Evolving Language Among Agents

by

AnYuan Guo

Submitted to the Department of Electrical Engineering and Computer Science
in partial fulfillment of the requirements for the degree of

Master of Engineering in Electrical Engineering and Computer Science

at the

MASSACHUSETTS INSTITUTE OF TECHNOLOGY

September 1999

Author .....
Department of Electrical Engineering and Computer Science
August 23, 1999

Certified by .....
Gerald J. Sussman
Matsushita Professor of Electrical Engineering
Thesis Supervisor

Accepted by .....
Arthur C. Smith
Chairman, Department Committee on Graduate Students

Evolving Language Among Agents

by

AnYuan Guo

Submitted to the Department of Electrical Engineering and Computer Science
on August 23, 1999, in partial fulfillment of the
requirements for the degree of
Master of Engineering in Electrical Engineering and Computer Science

Abstract

Agent-based systems are composed of autonomous entities(agents) that are responsive, pro-active and adaptive. Agents often cooperate with other agents in their environment to accomplish goals. Traditionally, agents communicate through pre-defined languages crafted by human agent designers. I have created a system that allows a class of agents to develop their own language. This class is composed of agents that needs to convey information intensive data exhibiting underlying regularities to other agents (such as a robotic agent communicating its complex sensor data). The functions pertaining to language development are encapsulated into a module called the Language Evolving Module (LEM) that can be embedded into an agent. A skeletal system has been implemented to demonstrate the evolution of language among a set of agents.

Thesis Supervisor: Gerald J. Sussman
Title: Matsushita Professor of Electrical Engineering

Acknowledgments

I would like to thank all these people, without whom this thesis would not have been possible:

- Prof. Marvin Minsky, whose book *The Society of Mind* (and the class he taught with the same name) has inspired me to pursue a career in AI research.
- Prof. Patrick Winston for offering a wonderful class, the Human Intelligence Enterprise from which I became acquainted with works that led to this thesis, and discussions with whom has shaped my thoughts on AI.
- My thesis advisor, Prof. Gerald Sussman, for providing many initial ideas for this thesis, and for his guidance, inspiration and encouragement.
- My mom, Ma Yi-Tian, and my dad, Guo Jing-Wu, for the many sacrifices they made in immigrating to the States so my brother and I could get a better education.
- My grandmother, Guo Yu-Zhen, for having raised me in the absence of my parents and for the influences on my personal development.
- My brother, Guo Dao-Yuan, for his advise and support.
- Rick Rikoski, Eugene Kang, Dan Omura and Hirsh Nanda for proof reading my thesis.
- Hirsh Nanda, for the discussions that clarified my ideas, for all your love, support, and therapeutic effects.

Contents

Chapter 1

Introduction

The term "agent" holds different meanings in different subfields of computer science. In the context of this thesis, we define "agent" to be an autonomous system situated in a dynamic environment. An agent is responsive, pro-active, and adaptive. The term "autonomy" refers to the agent's ability to act without the direct intervention by humans or by other agents. An agent should have control over its own actions and internal states. It must not only be able to respond to changes in its environment, but should also be pro-active, exhibit opportunistic and goal-directed behavior, and learn from its interactions with the environment [5].

An agent-based system can be purely software. For example, OASIS is an agent-based system that specializes in air traffic control. Agents are used to represent both aircraft and the various air traffic control systems in operation. An agent can be instantiated with the information and goals corresponding to the real-world aircraft, vying for resources such as runway space [5].

An agent can also be physically embodied. The MIT Mobile Robot Group is building small robot rovers to explore the terrains of Mars. They are designed to work in colonies to collect rock samples, take pictures and prepare the site for manned landing [1].

An agent could also be "situated" in the human mind. In *The Society of Mind*, Marvin Minsky presents a theory of the mind as a society of small components or agents, each of which is designed to carry out one or two simple tasks [8]. Intelligence

emerges when these agents interact with one another.

In an agent-based system, whether the agents are pieces of software, physically embodied robots, or cognitive abstractions, each is situated in an environment that includes many other agents with which it cooperates to achieve its goals. In order to communicate successfully, they must agree on several levels [3]:

- Transport: how agents send and receive messages;
- Language: what the individual messages mean;
- Policy: how agents structure conversations;
- Architecture: how to connect systems in accordance with constituent protocols.

This thesis is concerned with the language level.

The class of agent that this dissertation is concerned with consists of those that needs to communicate information intensive data to other agents. More specifically, that data should hold underlying regularities. For example, a robotic agent perceives its physical environment through its sensors. It may need to communicate sensory data to other agents. In this case, a language that can abstract that information can be used in communication.

Agents usually communicate through a pre-defined language crafted by programmers, with set syntax and semantics. The goal of this project is to explore the possibility for agents to develop their own language.

1.1 Overview

- Section 2 discusses motivation for the project.
- Section 3 provides relevant background information. It describes the natural language learning process that this system draws from.
- Section 4 describes each component of the Language Evolving Module.

- Section 5 describes an example system that implements the Language Evolving Module and presents the results of the experiment.
- Section 6 draws conclusions and discusses future work.

Chapter 2

Motivation

The benefits of endowing the agents with the ability to develop their own language span several levels, including agent application, program design, software engineering, and philosophical level.

From the view of agents trying to achieve their goals, a human-crafted agent communication language may not be best suited to the task at hand. For instance, if the programmer does not fully anticipate the needs of the agents, projects will fail when agents encounter new situations about which they cannot communicate [11]. On the other hand, if agents have the ability to evolve their own language, they will possess the mechanisms to accommodate change by augmenting the language.

Due to the agent's ability to adapt the language according its to needs, the language does not have to be bogged down with the ability to anticipate every contingency; it can be smaller and more compact.

The main beneficiaries of this scheme are programmers and agent designers. In a system where agents evolve their own language, humans have to do much less development and maintainence.

From a software engineering perspective, agents with an embedded language development component are more modular. Under this paradigm, agents are made to be more general-purpose. They can be transferred from one task to another with minimal modification, promoting software and/or hardware reuse.

Most importantly, this development shifts decision making responsibility from the

programmer to the program itself. The end product is a smarter piece of software that is more robust, versatile and adaptable. These are the hallmarks of intelligence.

Chapter 3

Background

Much work has been done on the origin of language. Simon Kirby presented a particularly interesting approach [7]. He simulates the evolution of syntax from initially language-less individuals. He argues that compositionality, a particular syntactic property of natural language, can be evolved as an outcome of the dynamics of an observationally learned adaptive system.

In Kirby's simulation, two individuals communicate with each other: one designated as the speaker, the other as the learner. The speaker passes an utterance and its corresponding meaning to the learner. The learner incorporates the pair into its grammar and attempts to merge the grammar rules to produce a shorter rule set. This process of incorporating and merging utterances and meanings eventually leads to a compositional language with vocabulary for each meaning unit.

My model for the development of agent communication language draws from this work in natural language learning. This idea was suggested by Gerald Sussman and Kenneth Yip. I generalized Kirby's work by encapsulating the language development functions into a module that can be configured to work with different types of agents. This module will be embedded into every agent in an agent-based system. Before they are deployed into normal operation, agents first use the module to evolve a common language.

Chapter 4

Language Evolving Module

The system I propose differs radically from those which agents use to communicate today. Instead of using a language specified by the programmer, agents cooperate to evolve their own language. A language is a set of agreed-upon ways to express certain meanings. It is composed of a vocabulary and a set of rules to combine and manipulate the vocabulary to form more complex expressions.

The functions required for those tasks are encapsulated into a Language Evolving Module(LEM). It is embedded into each agent. Before agents go into normal operation, an initial setup time is required where agents pass around messages and the meanings those messages present. Each agent then processes messages received in their LEMs. By the end of the run, a common language is reached and agents only need to pass each other messages to successfully communicate.

To illustrate the purpose of the different components of the LEM, I will give a crude example from natural language learning that has the same flavor as the work done by an LEM.

During the language acquisition period, a child hears the messages "this is a table" and "this is a chair", and is shown a table and a chair respectively. Stimulated by the these two objects, two patterns of neuron activations form in the child's brain. The child picks out those two objects from their surroundings, and simultaneously notices the difference in the message is between "table" and "chair". The child then associates "table" with the set of neural patterns corresponding to that invoked by

the table, and does similarly for "chair".

In this process, the child has to be able to perform pattern recognition on the messages and on the meaning (represented as neuron activations). This function corresponds to the pattern recognition module in this system(described in section 4.5). The child associates differences in the messages with differences in the meanings; this is paralleled in the inducer(section 4.4). These associations have to be stored in some kind of structure; we use a grammar(section 4.1). The semantic information in the example is made up of neuron activation patterns. Semantic representation is discussed in section 4.3. Lastly, the message generator(section 4.2) builds messages that corresponds to those heard by the child.

4.1 Message-Meaning Association: Attribute Grammar

The LEM has a storage module that keeps all associations between messages and meanings. The association can be a simple one-to-one association, in which case an agent simply records messages and meanings that other agents send it. There are two problems with this scheme. First, if the meaning space is large, for every meaning a unique message is needed to represent it. The demand on memory storage will be intensive. Secondly, since different agents can invent different ways to express the same meaning, conflict resolution mechanisms will be needed in order to ensure consistency.

The implementation I chose captures the semantic relationship with a type of grammar called attribute grammar. Attribute grammar is a class of context-free grammar with simple semantic extensions. The semantics is represented as feature-value pairs.

4.2 Message Generator

Language maps messages onto meanings and vice versa. A message is composed of discrete units that can be transported between the parties involved in the exchange. In natural language, the messages are sentences made up of words. In an agent language, a message will also have to be composed of discrete units.

The message generator takes as its input a list of symbols that are specified by the agent designer. They can be 1's and 0's, letters of the alphabet, or user defined units. Each symbol can also have a probability of occurence attached to it.

Initially, during the language-evolving stage, random messages are generated obeying the distribution specified by those probabilities. Slowly, syntactic structure builds up in the language. Vocabulary that expresses meaning components are formed so that only part of the message has to be randomly invented. By the end of the development process, a coordinated language will emerge with set syntax and semantics. The message generator will deterministically generate messages that express desired meanings according to those syntactic and semantic rules.

4.3 Semantics

A language is used to express meanings. The meaning space in human communication is anything that we can experience with our senses. It can be a set of physiological signals that we give names such as hunger, itch or pain. It can be visual signals that we perceive as color, intensity, or dynamics. It can be sounds, tactile information, or patterns that are yet unclassified.

In the class of agents we are concerned with, meanings can range from high level data such as program outputs to low level data such as sensor measurements(deceleration, light intensity, etc).

In the LEM, the semantic format that the LEM works with is feature-value pairs. For example, if the meanings represented are data taken from a digital camera, then the features could be pixel indices and the values could be the RGB values at those

indices.

Why do agents need to form a language in the first place? Why don't they just pass each other raw data such as sensor values? There are two reasons: reduced bandwidth consumption and increased modularity. If agent A takes in data from a camera and agent B needs to know what agent A sees, agent A can either send all its sensor data. or it can describe what it sees in a sentence or two using a language which agent B understands. The second option imposes much lower demands on bandwidth.

Another desirable outcome of communicating with a language rather than passing raw semantic data is increased modularity. The function that interprets the data is encapsulated in the object that serves the data. Processing is done locally. In this manner, if a function needs to be modified, only one copy is affected, as opposed to multiple copies in the alternative scheme of embedding functions into every agent that requires the data.

4.4 Inducer

The purpose of the inducer is to associate the patterns in the meaning with the patterns in the message string. It attempts to find the most general syntactic rule to express meanings in the semantic domain. This scheme is based on the view that learning is the compression of observed data into a compact hypothesis [9]. Each agent builds a grammar through inducing the message-meaning pairs produced by other agents in the system.

The induction algorithm used was developed by Stolcke [10] and further refined by Kirby [6]. The learning process involves two steps: incorporation and merging.

Incorporation is the process by which agents build a simple grammatical model for each message-meaning pair received and add that model to the pool of existing ones.

Merging involves modifying the grammatical models in such a way that they become more similar. Redundant models are deleted from the pool. In this way, the

models become more general; in other words, they are able to express a larger number of meanings.

4.5 Pattern Recognition Functions

The function of the pattern recognizer is to pick out the relevant patterns in the meaning data. These functions are supplied by the agent designers.

The idea is to take advantage of the regularities in the meanings. By naming those regularities, the amount of information that needs to be sent is much lower by communicating with the language rather than with the original data.

Two sets of pattern recognition functions are needed, one to pick out patterns in the message, the other to pick out patterns in the meaning.

Pattern recognition functions for the expression should be kept simple because they determine the syntactic complexity of the language developed. According to the current implementation of the inducer, the pattern recognition function can be as simple as finding two differences in the expressions. For example, if one expression is "0101000" and the other is "010001", then by simple string matching, we can pick out the two chunks of differences "1000" and "001".

After we pick out the differences in the expressions, we examine their corresponding meanings. Two functions are needed. One will determine whether two meanings can be considered the same, the other will pick out the chunks where they differ. The pattern recognition functions for meanings can be much more complicated because that is where much of the program's intelligence lie. The resolution at which we contrast two sets of meanings can determine what kind of patterns are named in the language.

One motivation for this project is to grant the agents more autonomy by shifting the responsibility of language design from programmers onto the agents. Asking the agent designers to provide these pattern recognition functions seems to compromise this goal to some extent. However, I believe that by encapsulating these functions into a module, we can better utilize a vast amount of pattern recognition research.

Chapter 5

An Example System

5.1 Implementation of the specific modules

We will examine in detail a specific example of an agent evolution experiment implemented using the LEM prototype described in the previous chapter.

The expressions that KGB agents will utter to each other are strings consisting of the letters a, b, c, d and e. The meanings are an abridged version of a visual image taken with a camera. It is a square composed of 4 pixels. We index them by their vertical and horizontal position. For example, the top left pixel would be (1,1). The values of each pixel can be either white or black, except for the bottom right pixel(2,2) which is always set to black. This is an attempt at modeling regularities in the semantic space.

The semantics is represented in a feature value format. An example meaning could be:

(1,1) = black
(1,2) = white
(2,1) = black
(2,2) = black

The pattern recognition function for expressions simply compares two expressions and tries to see whether the two expressions differ by a pair of substrings. For example,

the expressions "aabddd" and "aabeea" differ by "ddd" and "eea".

Two pattern recognition functions for meanings are needed, one which will be able to tell whether two meanings are the same, and another to pick out chunks of differences in two meanings. Naive versions of these two functions are implemented. The similarity function checks for strict equivalence. The difference function returns two groups of feature value pairs that differ between the two meanings.

The induction engine uses two heuristics. One of them assumes that if two meanings are the same, and if the two expressions denoting those meanings differ only in one place, the difference in expressions can be erased. One rule will be used to generate both expressions. The other induction heuristic attributes the difference in the expressions to the differences in the meanings.

5.2 Simulation Cycle

Six agents are connected in a ring formation, each having two neighbors. They are picked randomly to initiate communication. The one chosen will select a meaning to express, if it can express that meaning, it will pass the message and the meaning as a pair to all its neighbors. If not, it will most likely remain silent, but once in a while, use the message generator to invent a random expression to express that meaning. If an agent has several ways to express the same meaning, then it will use the most general grammar rule - the one that is able to express the greatest number of meanings.

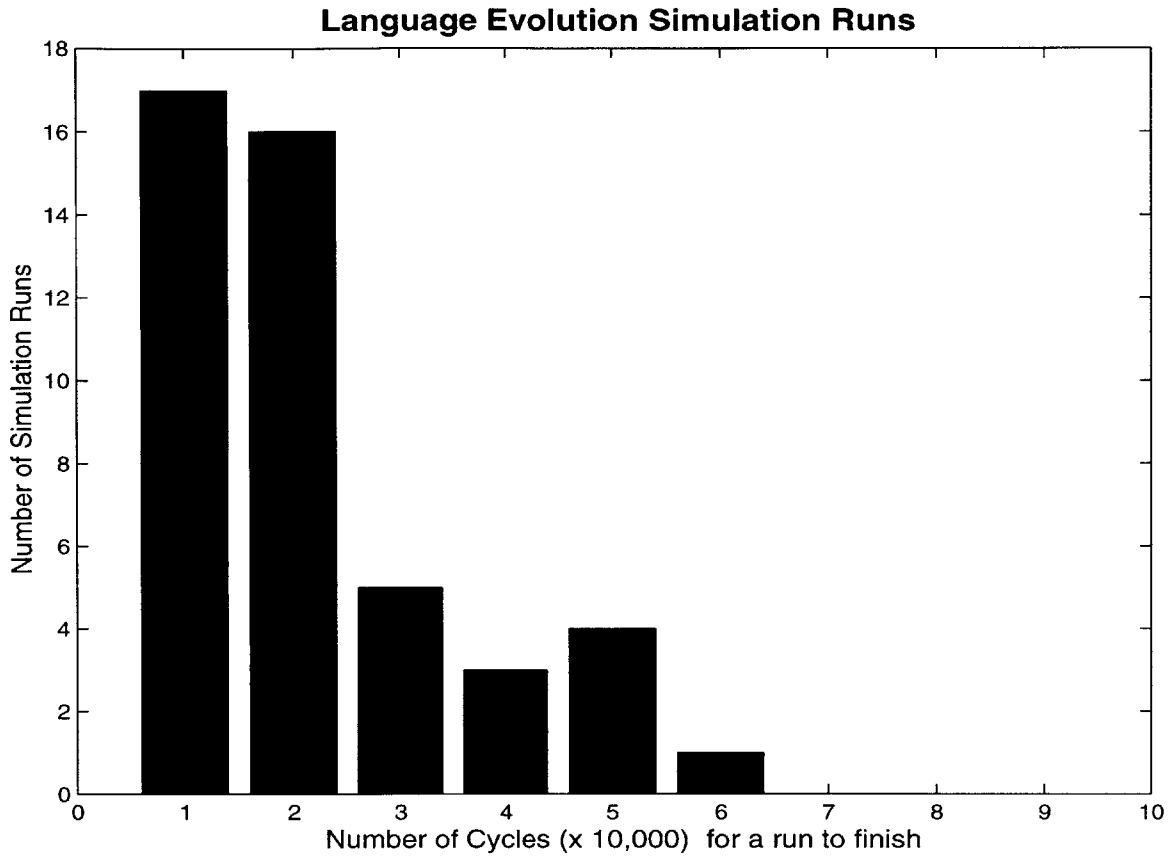
Each agent periodically erases its grammar structure in the hope of learning more advanced structures from its neighbors as the simulation progresses.

5.3 Results

Over 50 runs of the simulation, the agents converge to the same language that is fully compositional(only one grammar rule is needed to express all meanings in the meanings space) after an average of 14,872 cycles, where a cycle is defined as one

agent initiating and completing communication with its neighbors.

Of the 50 runs, 4 runs did not converge within 100,000 cycles. The convergence statistics for the rest of the 46 runs is shown in the graph below.



A sample grammar of an agent in the beginning of a simulation is very idiosyncratic(random strings represent each meaning) and lacks any compositional structure:

S – a c c d
  11 = black
  12 = white
  21 = black
  22 = black

S – d b b c a
  11 = white

12 = white
21 = black
22 = black

S – c b b d c c
11 = black
12 = white
21 = white
22 = black

S – b c a c a d
11 = white
12 = white
21 = white
22 = black

By the end of the simulation, the grammar is fully compositional. Vocabulary and syntax are formed.

S – 7 d 6
22 = black
More feature-values pairs in: (6 7)

7 – 4
12 = black
More feature-value pairs in: (4)

4 – a b c d c a
21 = white

4 – a c d c c b
21 = black

6 – e d
11 = white

6 – b a b a b
11 = black

Notice that the agent had invented different ways to express the same meaning. The words "a b c d c a" and "e d" are synonyms for "white". Word order and vocabuary for meaning-components are formed by the end of the simulation.

Chapter 6

Conclusion

The simulation I have implemented demonstrates that agents can indeed evolve their own language. Right now, the simulation is rather slow. In order to scale it up to handle more realistic applications, optimizations will have to be made to the program.

Many interesting questions still remain. For example, the agent designers are asked to provide the pattern recognition functions in the Language Evolving Module. It would be interesting to explore the possibility of letting agents learn to recognize useful patterns on their own, possibly through reinforcement learning.

In this dissertation, I have defined and implemented a Language Evolving Module that can be embedded into agents to allow them to develop a common language. I have shown how this system can be used to create a compact language for agents to communicate information-intensive data that exhibit underlying regularities.

Bibliography

[1] Rodney Brooks. Fast, cheap and out of control: A robot invasion of the solar system. *Journal of the British Interplanetary Society*, pages 478–485, October 1989.

[2] Ken Dickey. substr.scm. 1991.

[3] Tim Finin. Specification of the kqml agent-communication language. Technical report, The DARPA Knowledge Sharing Initiative External Interfaces Working Group, 1993.

[4] Gerald Jay Sussman Hal Abelson and Julie Sussman. *Structure and Interpretation of Computer Programs.* MIT Press and McGraw-Hill, 1985.

[5] N.R. Jennings and M.J. Wooldridge. *Agent Technology: Foundations, Applications, and Market.* Springer-Verlag, 1985.

[6] Simon Kirby. *Fitness and the selective adaptation of language. In Approaches to the Evolution of Language: Social and Cognitive bases.* Cambridge University Press, 1997.

[7] Simon Kirby. Syntax without natural selection: how compositionality emerges from vocabulary in a population of learners. Technical report, Department of Linguistics, University of Edinburgh, 1999.

[8] Marvin Minsky. *The Society of Mind.* Simon and Schuster, 1988.

[9] Miles Osborne and Ted Briscoe. Learning stochastic categorial grammars. Technical report, Computer Laboratory, Cambridge University, 1987.

[10] Andreas Stolcke. *Bayesian Learning of Probabilistic Language Models.* PhD thesis, University of California at Berkeley, 1994.

[11] Holly Yanco. Robot communication: issues and implementations. Master's thesis, Massachusetts Institute of Technology, 1994.

Appendix A

Source Code

A.1 Simulation

```
(declare (usual-integrations))

;; Parameters needed for initializing the simulation
;; ------------------------------------------------
(define *max-cycles-per-sim* 100000)
(define *num-of-runs* 200)


;; Parameters needed for initializing the agent-set
;; ------------------------------------------------
;; *num-of-agents*  -  The number of agents in the agent set
;;
;; *max-rounds* - The maximum number of rounds that induction
;;                 will be performed on a rule-set after
;;                 incorporating a new message-meaning pair.
;;                Induction almost always done before this
;;                  number is reached. It is installed to
;;                  more for debugging purposes to ensure that
;;                  no inifinite loop ties up the simulation.
;;
;; *invention-chance* - For 1 out of *invention-chance* number of
;;                times, an agent will invent a message for a meaning
;;                it dosen't know how to express.
;;
```

```
;; *life-span* -  The number of rounds after an agent is prompted to
;;                speak an agent is cleared, unless the agent has
;;                reached convergence.
;; ------------------------------------------------------------------
(define *num-of-agents* 6)
(define *max-rounds* 40)
(define *invention-chance* 60) ;1 in n
(define *life-span* 40)


;; Parameters needed for initializing each agent
;; ---------------------------------------------
;; *utterance-list*  -  The utterance-list
;; *min-chunk%*  - In order for a substring to be chunked, it has
;;                 to be longer than
;;                 the this percent of the right-hand-side rule length.
;; *max-chunk%*  - In order for a substring to be chunked, it has to
;;                 be shorter than
;;                 the this percent of the right-hand-side rule length.
;; *min-utterance-length*  -  The shortest possible length for an
;;                            invented utterance.
;; *min-utterance-length*  -  The longest possible length for an
;;                            invented utterance.
;; *stop-length-for-chunking*  - When the rhs reaches this length,
;;                               chunking no longer
;;                               happens to this rule.
;; *feature-values-ls* - Lists the values that each feature can
;;                       possibly take on.


(define *utterance-list* '((a 2) (b 4) (c 3) (d 2) (e 1)))
(define *min-chunk%* 0.15)
(define *max-chunk%* 0.8)
(define *min-utterance-length* 4)
(define *max-utterance-length* 8)
(define *stop-length-for-chunking* 4)
(define *non-terminals* '(a b c d e))
(define *feature-values-ls*
  '((11 (white black))
    (12 (white black))
    (21 (white black))
    (22 (black))))

;; This simulation contains 6 agents connected in a ring.
;; The maximum number of cycles that will be run is 100,000,
```

```
;; this is so that the simulation doesn't get stuck on the
;; extreme case. Most cases finish well within that.

(define (simulation-loop iteration-num)

  (define (simulation n)

    (if (equal? 0 (modulo n 100))
(begin
  (display "***************** Simulation cycle # ")
  (display n)(newline)))

    (if (equal? 0 (modulo n 500))
(begin
  (for-each (lambda (agent)
      (let ((num-generable
    (agent 'num-of-generable-meanings))
    (num-needed (length (agent 'rules-needed))))
(display " (")
(display num-generable)
(display " ")
(display num-needed)
(display ")")))
    (agent-set 'agent-set->list))
  (newline)))

    (if (and (equal? 0 (modulo n 100))
     (for-all?
 (agent-set 'agent-set->list)
       (lambda (agent)
 (let ((num-generable
                       (agent 'num-of-generable-meanings))
       (num-needed (length (agent 'rules-needed))))
   (and (= num-generable (agent 'total-num-of-meanings))
(= num-needed 1))))))
(simulation-loop (+ num-iterations 1)))

    (if (> n *max-cycles-per-sim*)
(if (> (+ iteration-num 1) *num-of-runs*)
    'done
    (simulation-loop (+ iteration-num 1)))
(let ((id (+ 1 (random *num-of-agents*))))
  ((agent-set 'apply-action) id)
  (simulation (+ n 1)))))
```

```
  (newline)
  (display "------------------------ Run # ")
  (display iteration-num)
  (display "------------------------")
  (newline)

  (define agent-set
     (make-ring-configuration-agent-set *num-of-agents*))
  (simulation 1))

(simulation-loop 1)
```

A.2 Agent Set

```
(declare (usual-integrations))

;; Creates an agent set where the agents are connected in a
;; ring configuration.


;; Parameters needed for initializing the agent-set
;; ------------------------------------------------
;; *num-of-agents*  -  The number of agents in the agent set
;;
;; *max-rounds* - The maximum number of rounds that induction
;;                 will be performed on a rule-set after
;;                 incorporating a new message-meaning pair.
;;                Induction almost always done before this
;;                  number is reached. It is installed to
;;                  more for debugging purposes to ensure that
;;                  no inifinite loop ties up the simulation.
;;
;; *invention-chance* - For 1 out of *invention-chance* number of
;;                times, an agent will invent a message for a meaning
;;                it dosen't know how to express.
;;
;; *life-span* -  The number of rounds after an agent is prompted to
;;                speak an agent is cleared, unless the agent
;;                has reached convergence.
;; ------------------------------------------------------------------

(define (make-ring-configuration-agent-set num-of-agents)
  (let ((agent-set (make-agent-set *max-rounds*
   *invention-chance*
   *life-span*)))
    (define (init-loop i)
      (if (> i num-of-agents)
  'done-initializing-agent-set
  (let* ((id (agent-set 'get-new-id))
 (neighbors
  (cond ((equal? 1 num-of-agents)
 '())
((and (equal? 2 num-of-agents)
      (equal? 1 i))
 '(2))
((and (equal? 2 num-of-agents)
```

```
      (equal? 2 i))
 '(1))
((equal? id 1)
 (list num-of-agents (+ id 1)))
((equal? id num-of-agents)
 (list 1 (- id 1)))
(else (list (- id 1) (+ id 1)))))
 (a (make-agent id
neighbors
*utterance-list*
*min-chunk%*
*max-chunk%*
*min-utterance-length*
*max-utterance-length*
*stop-length-for-chunking*
*non-terminals*
*feature-values-ls*)))
    (if (odd? i)
((a 'set-age!) 300))
    ((agent-set 'add-agent!) a)
    (init-loop (+ i 1)))))
    (init-loop 1)
    agent-set))

;; Creates an angent set where every agent is connected to every other
;; agent in the set.

(define (make-fully-connected-agent-set num-of-agents . initial-id-num)
  (let* ((agent-set
(make-agent-set *max-rounds* *invention-chance* *life-span*))
 (min-id (if (null? initial-id-num)
     1
     (car initial-id-num)))
 (max-id (if (null? initial-id-num)
     num-of-agents
     (+ (car initial-id-num) (- num-of-agents 1))))
 (id-list (fill-list min-id max-id)))

    (define (init-loop i)
      (if (> i max-id)
  'done-initializing-agent-set
  (let* ((id (agent-set 'get-new-id))
 (neighbors (delq id id-list)))
    ((agent-set 'add-agent!)
     (make-agent id
```

```
 neighbors
 *utterance-list*
 *min-chunk%*
 *max-chunk%*
 *min-utterance-length*
 *max-utterance-length*
 *stop-length-for-chunking*
 *non-terminals*
 *feature-values-ls*))
    (init-loop (+ i 1)))))
    ((agent-set 'set-current-id) min-id)
    (init-loop min-id)
    agent-set))


;; The base agent-set manipulation functions.

(define (make-agent-set %max-rounds %invention-chance %life-span)
  (let ((%ls '())
(%current-id 1)
(%non-converged-agents '()))

    (define (clear)
      (begin (set! %ls '())
     (set! %current-id 0)
     (set! %non-converged-agents '())
     unspecific))

    ;; id starts at 1
    (define (get-new-id)
      %current-id
      (set! %current-id (+ %current-id 1)))

    (define (set-current-id! id)
      (begin
(set! %current-id id)
unspecific))

    (define (non-converged-agents)
      %non-converged-agents)

    (define (remove-from-non-converged-agents id)
      (set! %non-converged-agents
    (delq id %non-converged-agents)))
```

```
    (define (is-a-non-converged-agent id)
      (member id %non-converged-agents))

    (define (empty?)
      (null? %ls))

    (define (get-agent id)
      (list-search-positive
  %ls
(lambda (agent)
  (equal? id (agent 'id)))))

    (define (add-agent! agent)
      (begin
(set! %ls (append %ls (list agent)))
(set! %non-converged-agents
      (cons (agent 'id) %non-converged-agents))
unspecific))

    (define (delete-agent! agent)
      (define (same-id? a)
        (equal? (agent 'id) (a 'id)))
      (let ((new-agent-set ((list-deletor same-id?) %ls)))
(begin
  (set! %ls new-agent-set)
  (set! %non-converged-agents
(delq (agent 'id) %non-converged-agents))
  unspecific)))

    (define (agent-set->list)
      %ls)

    ;; returns a list of agents objects that are neighbors
    ;; of the agent arg
    (define (find-neighbors agent)
      (let ((neighbor-ids (agent 'neighbors)))
(map (lambda (id)
       (let ((a (get-agent id)))
 (if (null? a)
     (error "In find-neighbors: no agent with id: " id)
     a)))
     neighbor-ids)))

    (define (apply-action id)
      (let ((the-agent (get-agent id)))
```

```
(if (null? the-agent)
    (error
      "agent with id: " id " does not exist -- apply-action"))
(let* ((fv-ls (the-agent 'generate-meaning))
       (utt-inventor (the-agent 'utterance-inventor))
       (neighbors (find-neighbors the-agent)))

  (the-agent 'increment-age!)

           ;; the-agent will only receive actions for %life-span number
   ;; of times. After that, a new blank agent goes to work.
   ;; unless that agent has converged already. In which case,
   ;; that agent will not be cleared again.

   ;; if the agent has already converged. Then it just talks to
   ;; the neighbors
  (cond ((not (is-a-non-converged-agent (the-agent 'id)))
 (let ((utt ((utt-inventor 'invent) fv-ls)))
   (for-each
    (lambda (ag)
      (if (not (((ag 'utterance-inventor) 'can-parse?)
utt
fv-ls))
  (begin ((ag 'incorporate) utt fv-ls)
 ((ag 'induce) %max-rounds))))
    neighbors)))

;; if the agent is older than %life-span,
        ;; then, if it converged,
;; remove it from the non-convg-agents list
;; if the old agent hasn't converged, we clear it.
((> (the-agent 'age) %life-span)
 (if (the-agent 'has-converged?)
     (remove-from-non-converged-agents (the-agent 'id))
     (begin (the-agent 'clear!)
    (display "agent #")
    (display id)
    (display " is cleared...")
    (newline))))


;; If the agent is not old, and is not off of the
        ;; con-convg-agents list,
;; the invent function of the utterance inventor will
;; first check to see whether that meaning can be
```

```
;; expressed. If so, fill out all the non-terminals
;; in that rule's rhs and
;; return that utterance.
;; If it can be partially generated, randomly generate
  ;; the rest.
;; Else, give a random string.
(else
 (if (or ((utt-inventor 'can-generate-meaning?) fv-ls)
 (one-in-n-chance %invention-chance))
     (let ((utt ((utt-inventor 'invent) fv-ls)))
       (for-each
(lambda (ag)
  (if (not (((ag 'utterance-inventor) 'can-parse?)
    utt fv-ls))
      (begin ((ag 'incorporate) utt fv-ls)
     ((ag 'induce) %max-rounds))))
(append (list the-agent) neighbors)))))))))

    (define (print)
      (for-each
       (lambda (agent)
 (agent 'print))
       %ls))

    (define (dispatch m)
      (cond ((eq? m 'clear) (clear))
    ((eq? m 'set-current-id!) set-current-id)
    ((eq? m 'get-new-id) (get-new-id))
    ((eq? m 'empty?) (empty?))
    ((eq? m 'get-agent) get-agent)
    ((eq? m 'add-agent!) add-agent!)
    ((eq? m 'non-converged-agents) (non-converged-agents))
    ((eq? m 'is-a-non-converged-agent) is-a-non-converged-agent)
    ((eq? m 'remove-from-non-converged-agents)
     remove-from-non-converged-agents)
    ((eq? m 'delete-agent!) delete-agent!)
    ((eq? m 'agent-set->list) (agent-set->list))
    ((eq? m 'find-neighbors) find-neighbors)
    ((eq? m 'apply-action) apply-action)
    ((eq? m 'print) (print))
    (else
      (error "Unknown operation in MAKE-AGENT-SET -- " m))))
   dispatch))
```

A.3 Agent

```
(declare (usual-integrations))

(define (make-agent %id
    %neighbors
    %utterance-list
    %min-chunk%
    %max-chunk%
    %min-utterance-length
    %max-utterance-length
    %stop-length-for-chunking
    %non-terminals
    %feature-values-ls)

  (let* ((%id id)
 (%neighbors neighbors)
 (%age 0)
 (%utterance-list (list (list 'a 2)
(list 'b 4)
(list 'c 3)
(list 'd 2)
(list 'e 1)))
 (%min-chunk% 0.15)
 (%max-chunk% 0.8)
 (%min-utterance-length 4)
 (%max-utterance-length 8)
 (%stop-length-for-chunking 4)
 (%non-terminals '(a b c d e))
 (%feature-values-ls '((11 (white black))
       (12 (white black))
       (21 (white black))
       (22 (white))))
 (%grammar (make-grammar %non-terminals '() (make-rule-set)))
 (%induction-engine (make-heuristic-induction-engine
     %grammar
     %min-chunk%
     %max-chunk%
     %stop-length-for-chunking))
 (%utterance-inventor
   (make-utterance-inventor %utterance-list
               %grammar
    %min-utterance-length
    %max-utterance-length))
```

```
 (%meaning-generator
  (make-random-fv-generator %feature-values-ls))
 (%convg-flag 0))

    (define (incorporate utt m-ls)
      ((%induction-engine 'incorporate) utt m-ls))

    ; the meaning returned is a fv list
    (define (generate-meaning)
      (%meaning-generator 'get-fv-list))

    ;; returns an utterance
    (define (talk fv-ls)
      (let* ((fvs-meaning (make-fv-group
   (init fv-ls))))
(%utterance-inventor fvs-meaning)))

    (define (induce n)
      ((%induction-engine 'induce) n))

    (define (id)
      %id)

    (define (grammar)
      %grammar)

    (define (neighbors)
      %neighbors)

    (define (age)
      %age)

    (define (set-age! age)
      (begin
(set! %age age)
unspecific))

    (define (convg-flag)
      %convg-flag)

    (define (clear-convg-flag)
      (begin
(set! %convg-flag 0)
unspecific))
```

```
    (define (incr-convg-flag!)
      (begin
(set! %convg-flag (+ %convg-flag 1))
unspecific))

    (define (increment-age!)
      (begin
(set! %age (+ %age 1))
unspecific))

    (define (utterance-inventor)
      %utterance-inventor)

    (define (meaning-generator)
      %meaning-generator)

    (define (induction-engine)
      %induction-engine)

    (define (clear!)
      (begin
(%grammar 'clear)
(set! %age 0)
(set! %convg-flag 0)
unspecific))

    ;; the number of possible combinations of meaning
    ;; as defined in %feature-values-ls
    (define (total-#-meanings)
      ;; ls is of the form ((a b c) (y z))
      ;; 3 * 2 = 6 gives the num of possible combinations
      (define (total-aux ls count)
(if (null? ls)
    count
    (total-aux (cdr ls)
       (* (length (car ls)) count))))
      (let ((values-ls (map (lambda (fv)
      (second fv))
    %feature-values-ls)))
(total-aux values-ls 1)))

    ;; construct a list of all possible meanings generable
    ;; from the %utterance-ls
    (define (list-generable-meanings)
```

```
      ;; add parens takes a list of lists of elements and
      ;; adds () around those elements
      ;; ((a b) (d e f)) -> (((a) (b)) ((d) (e) (f)))
      ;; it's easier to convert the input to this format than
      ;; it is to change the definition of make-combinations.
      (define (add-parens ls)
(define (add-aux the-ls result)
  (if (null? the-ls)
      result
      (add-aux (cdr the-ls)
       (append
result
(list (map (lambda (x)
     (list x))
   (car the-ls)))))))
(add-aux ls '()))


      ;; takes (agent action pred) ((mary loves john) (pete likes mary) ..)
      ;; -> (((agent mary) (action loves) (pred john))
      ;;     ((agent pete) (action likes) (pred mary)))
      (define (combine-feat-val features values-ls)
;; combine: (1 2 3) (x y z) -> ((1 x) (2 y) (3 z))
(define (combine ls1 ls2)
  (if (or (null? ls1)
  (null? ls2))
      '()
      (cons (list (car ls1)
  (car ls2))
    (combine (cdr ls1) (cdr ls2)))))
(map (lambda (vals)
       (combine features vals))
     values-ls))

      (let* ((features (map (lambda (fv)
      (first fv))
    %feature-values-ls))
     (possible-values (map (lambda (fv)
     (second fv))
   %feature-values-ls))
     (value-lists
      (make-combinations (add-parens possible-values))))
(combine-feat-val features value-lists)))

    (define (num-of-generable-meanings)
```

```
      (define (generable-aux ls count)
(cond ((null? ls) count)
      (((%utterance-inventor 'can-generate-meaning?)
(car ls))
       (generable-aux (cdr ls) (+ count 1)))
      (else (generable-aux (cdr ls) count))))
      (let ((possible-fv-lists (list-generable-meanings)))
(generable-aux possible-fv-lists 0)))

    ;; return the list of rules needed to generate all the possible
    ;; meanings in the %utterance-list
    (define (rules-needed)
      (define is-member? (member-procedure
  (lambda (r1 r2)
    (equal? (r1 'id) (r2 'id)))))
      (define (rules-needed-aux meanings the-rules)
(cond ((null? meanings) the-rules)
      (((%utterance-inventor 'can-generate-meaning?)
(car meanings))
       (let ((r ((%utterance-inventor 'pick-max-match-rule)
 (make-lhs '(S))
 (make-fv-group (init (car meanings))))))
 (if (is-member? r the-rules)
     (rules-needed-aux (cdr meanings) the-rules)
     (rules-needed-aux (cdr meanings)
       (cons r the-rules)))))
      (else (rules-needed-aux (cdr meanings) the-rules))))
      (rules-needed-aux (list-generable-meanings) '()))

    (define (has-converged?)
      (and (equal? (num-of-generable-meanings)
   (total-#-meanings))
   (equal? 1 (length (rules-needed)))))

    (define (print)
      (newline)
      (display "******************************")
      (newline)
      (display "Agent id: ")
      (display %id)(newline)
      (display "neighbors: ")
      (display %neighbors)(newline)
      (%grammar 'print)
      (newline))
```

```
(define (dispatch m)
  (cond ((eq? m 'incorporate) incorporate)
((eq? m 'generate-meaning) (generate-meaning))
((eq? m 'talk) talk)
((eq? m 'induce) induce)
((eq? m 'id) (id))
((eq? m 'grammar) (grammar))
((eq? m 'neighbors) (neighbors))
((eq? m 'age) (age))
((eq? m 'set-age!) set-age!)
((eq? m 'convg-flag) (convg-flag))
((eq? m 'clear-convg-flag) (clear-convg-flag))
((eq? m 'incr-convg-flag!) (incr-convg-flag!))
((eq? m 'increment-age!) (increment-age!))
((eq? m 'utterance-inventor) (utterance-inventor))
((eq? m 'meaning-generator) (meaning-generator))
((eq? m 'induction-engine) (induction-engine))
((eq? m 'total-num-of-meanings) (total-#-meanings))
((eq? m 'clear!) (clear!))
((eq? m 'list-generable-meanings) (list-generable-meanings))
((eq? m 'num-of-generable-meanings)
 (num-of-generable-meanings))
((eq? m 'rules-needed) (rules-needed))
((eq? m 'has-converged?) (has-converged?))
((eq? m 'print) (print))
(else
 (error
  "Unknown operation in AGENT -- " m))))
dispatch))
```

A.4 Inducer

```
(declare (usual-integrations))

;; The induction engine runs two heuristics on the rule set
;; in attemp to make the rules more similar so that redundant
;; rules can be deleted, leading to a small rule set that is
;; able to generate a large number of meanings.
;;
;; The two heuristics are Merge and Chunk. They are developed
;; by Andreas Stolcke and modified by Simon Kirby.
;;
;; Here, I implement a version that is described in Kirby's paper.
;; (See bibliography)
;;

(define (make-heuristic-induction-engine
 %grammar
 %min-chunk%
 %max-chunk%
 %stop-length-for-chunking)

  (define (grammar)
    %grammar)

  (define (incorporate utterance fv-ls)
    ((%grammar 'add-rule!)
     (make-rule (%grammar 'get-new-id)
(make-lhs (list 'S))
(make-rhs utterance)
(make-meaning
 (make-fv-group (init fv-ls))))))


  ;; run induction loop for n iterations or
  ;; until the grammar reaches fully induced state
  ;; (doesn't change any more)
  (define (induce n)
    (define (ind-loop i)
      (if (>= i n)
  (display "--------finish-inducing-n-rounds\n")
  (begin
    (run-mergeCat)
    ((%grammar 'rule-set) 'run-remove-duplicates!)
```

```
    (run-chunk)
    ((%grammar 'rule-set) 'run-remove-duplicates!)

    (if (%grammar 'rule-set-changed?)
(begin
  (%grammar 'clear-rule-set-change-marker!)
  (ind-loop (+ i 1)))
'done-inducing))))
    (%grammar 'clear-rule-set-change-marker!)
    (ind-loop 0))



  (define (run-mergeCat)
    (let ((id-list ((%grammar 'rule-set) 'valid-rule-ids)))
      (for-each
       (lambda (id1)
 (let ((r1 ((%grammar 'get-rule) id1)))
  (cond ((null? r1) unspecific)
 (else
  (for-each
   (lambda (id2)
     (let ((r1 ((%grammar 'get-rule) id1))
   (r2 ((%grammar 'get-rule) id2)))
       (cond ((>= id1 id2) unspecific)
     ((null? r1) unspecific)
     ((null? r2) unspecific)
     (else (mergeCat r1 r2)))))
   id-list)))))
       id-list)))


  (define (mergeCat r1 r2)
    (define (mergeCat? rule1 rule2)
      (let* ((lhs1 (r1 'lhs))
     (lhs2 (r2 'lhs))
     (rhs1 (r1 'rhs))
     (rhs2 (r2 'rhs))
     (diff-cats ((rhs1 'diff-one-pair-cats?) rhs2))
     (rule-set (%grammar 'rule-set)))
(cond ((and ((rhs1 'equal?) rhs2)
    (not ((lhs1 'equal?) lhs2))
    ((rule-set 'meaning-equal?) rule1 rule2))
       (list (car (lhs1 'lhs->list))
     (car (lhs2 'lhs->list))))
```

```
    ((and ((lhs1 'equal?) lhs2)
   diff-cats
   ((((r1 'meaning) 'fvs) 'equal?)
    ((r2 'meaning) 'fvs))
   ((rule-set 'meaning-equal?) rule1 rule2))
      diff-cats)
     (else #f))))
   (let ((cats (mergeCat? r1 r2)))
     (if (and cats
      ((%grammar 'is-non-terminal?) (first cats))
      ((%grammar 'is-non-terminal?) (second cats))
      (not ((%grammar 'cat-contains-cat?) (first cats)
  (second cats)))
      (not ((%grammar 'cat-contains-cat?) (second cats)
  (first cats))))
  (begin
    ((%grammar 'delete-rule!) r2)
    ((%grammar 'run-substitute-cat!) (second cats)
     (first cats))))))


 (define (run-chunk)
   (let ((id-list ((%grammar 'rule-set) 'valid-rule-ids)))
     (for-each
      (lambda (id1)
(let ((r1 ((%grammar 'get-rule) id1)))
  (cond ((null? r1) unspecific)
((<= ((r1 'rhs) 'length)
     %stop-length-for-chunking)
 unspecific)
(else
 (for-each
  (lambda (id2)
    (let ((r1 ((%grammar 'get-rule) id1))
  (r2 ((%grammar 'get-rule) id2)))
      (cond ((>= id1 id2) unspecific)
    ((null? r1) unspecific)
    ((null? r2) unspecific)
    ((not (((r1 'lhs) 'equal?) (r2 'lhs)))
     unspecific)
    (else
     (chunk-heuristics r1 r2)))))
  id-list)))))
      id-list)))
```

```
  (define (chunk-heuristics r1 r2)

    ;; Chunking first rule only:
    ;;
    ;; s -> a b c d d d
    ;;  (1,1) = black
    ;;  (1,2) = white
    ;;  (2,1) = black
    ;;
    ;; s -> a b c 1       where 1 -> e a a
    ;;  (1,1) = black          (1,2) = black
    ;;                         (2,1) = black
    ;;
    ;; results in:  adding 1 -> d d d
    ;;                       (1,2) = white
    ;;                       (2,1) = black
    ;; and deleting the first rule
    ;;
    (define (chunk-rule1? chunks
  rule1
  rule2)
      (let* ((fvs1 ((rule1 'meaning) 'fvs))
     (fvs2 ((rule2 'meaning) 'fvs))
     ; cdr of fvs-diff holds what fvs2(the shorter rule) lacks.
     (fvs-diff ((fvs1 'diff) fvs2)))
(if (null? chunks)
    #f
    (let* ((short-ck (second chunks))
   (long-ck (first chunks))
   (non-terms-in-long-ck
    (list-transform-positive
long-ck
      (lambda (cat)
((%grammar 'is-non-terminal?) cat)))))
      (if (and (equal? 1 (length short-ck))
       (> (length long-ck) 1)
       ((%grammar 'is-non-terminal?)
(car short-ck))
       ((fvs2 'strict-subset-of?) fvs1)
       (not (member (car short-ck) long-ck)))
  (cons long-ck
(make-meaning (make-fv-group (cdr fvs-diff))
      non-terms-in-long-ck))
```

```
 #f)))))

   (let* ((rhs1 (r1 'rhs))
  (rhs2 (r2 'rhs))
  (fvs&cat-diffs (meaning-diff-one-chunk? (%grammar 'rule-set)
  r1
  r2))
  (long-rhs-length (max (rhs1 'length) (rhs2 'length)))
  (short-rhs-length (min (rhs1 'length) (rhs2 'length)))
  (min-chunk-len (* long-rhs-length %min-chunk%))
  ;; We don't want to chunk anything that will take up
  ;; the entire rhs of the short rule.
  (max-chunk-len (min short-rhs-length
      (* long-rhs-length %max-chunk%)))
  (cks ((rhs1 'find-chunks) rhs2 min-chunk-len max-chunk-len))
  (cks1 ((rhs1 'find-chunks) rhs2 1 max-chunk-len))
  (cks2 (if (null? cks1)
    '()
    (cons (second cks1) (list (first cks1)))))
  (new-cat-name 'temporary)
  (chunk-r1-only (chunk-rule1? cks1 r1 r2))
  (chunk-r2-only (chunk-rule1? cks2 r2 r1)))
     (cond ((or (rhs1 'empty?) (rhs2 'empty?)) '())

   ((and cks fvs&cat-diffs)
    (let* ((fvs-lacks (car fvs&cat-diffs))
   (cat-lacks (cdr fvs&cat-diffs))
   (non-terms-in-ck1
    (list-transform-positive
(first cks)
      (lambda (cat)
 ((%grammar 'is-non-terminal?) cat))))
    (non-terms-in-ck2
(list-transform-positive
  (second cks)
 (lambda (cat)
  ((%grammar 'is-non-terminal?) cat))))
    (m1 (make-meaning (make-fv-group (cdr fvs-lacks))
      (cdr cat-lacks)))
    (m2 (make-meaning (make-fv-group (car fvs-lacks))
      (car cat-lacks))))
       (set! new-cat-name (%grammar 'get-new-cat-name))
       ((m1 'add-cats-if-lack!) non-terms-in-ck1)
       ((m2 'add-cats-if-lack!) non-terms-in-ck2)
       (chunk new-cat-name (first cks) m1)
```

```
      (chunk new-cat-name (second cks) m2)))

   (chunk-r1-only
    (let ((non-term (second cks1)))
      (chunk-first-rule-only r1
     r2
     non-term
     (car chunk-r1-only)
     (cdr chunk-r1-only))))
   (chunk-r2-only
    (let ((non-term (first cks1)))
      (chunk-first-rule-only r2
     r1
     non-term
     (car chunk-r2-only)
     (cdr chunk-r2-only)))))))


 ;; non-term is a list with a cat-name in it
 (define (chunk-first-rule-only r1 r2 non-term ck meaning)
   ((%grammar 'add-rule!) (make-rule (%grammar 'get-new-id)
     (make-lhs non-term)
     (make-rhs ck)
     meaning))
   ; this is not needed because we only chunk rules with the same lhs
   ; if this is added on later, add (let ((lhs1 (r1 'lhs) .. lhs2)))
   ;((%grammar 'run-substitute-cat!) (car (lhs1 'lhs->list))
   ;                                 (car (lhs2 'lhs->list)))
   ((%grammar 'delete-rule!) r1))


 (define (chunk cat-name cats meaning . add-rule?)
   (let ((lhs (make-lhs (list cat-name)))
 (rhs (make-rhs cats))
 (non-terms-in-cats (list-transform-positive
cats
      (lambda (cat)
((%grammar 'is-non-terminal?) cat))))
 (add? (if (null? add-rule?) #t (car add-rule?))))
     (if (null? cats)
 '()
 (begin
     (if add?
 ((%grammar 'add-rule!)
  (make-rule (%grammar 'get-new-id)
```

```
    lhs
    rhs
    meaning)))
    (for-each
     (lambda (r)
(let ((subchunk (meaning-is-strict-subchunk?
 meaning
 (r 'meaning)))
      (rhs2 (r 'rhs))
      (m (r 'meaning)))

 ; the first clause prevents recursion in cases where
 ; 792 -> a b b is added by the first chunk, second chunk
 ; adds 792 -> a, it substitutes the "a" in the first 792
 ; rule and makes it 792 -> 792 b b. Which is bad.

  (cond ((((r 'lhs) 'equal?) lhs) unspecific)
      subchunk)
 (begin ((rhs2 'substitute-cats!)
                                  cats
  (list cat-name))
((m 'delete-sub-meaning!) meaning)
((m 'delete-cats!) non-terms-in-cats)
((m 'add-cats!) (list cat-name)))))))
      (%grammar 'rule-set->list))))))

 (define (dispatch m)
   (cond ((eq? m 'grammar) (grammar))
 ((eq? m 'incorporate) incorporate)
 ((eq? m 'induce) induce)
 ((eq? m 'run-mergeCat) (run-mergeCat))
 ((eq? m 'run-chunk) (run-chunk))
 (else
  (error
   "Unknown operation in HEURISTIC INDUCTION ENGINE -- " m))))
 dispatch)
```

A.5 Message Generator

```
(declare (usual-integrations))

;; Given a meaning, the message inventor generates messages for that
;; meaning. There are three case:
;;
;;    1. If it can generate the meaning in full with the current grammar
;; structure. In this case, simply return the message that corresponds
;; to the meaning requested.
;;
;;    2. If it can generate part of the meaning, then invent the rest of
;; the meaning by randomly generating some strings.
;;
;;    3. If it cannot generate that meaning at all, generate a random
;; string using the Random-Message-Generator.
;;
;;
;; %message-list is of the form:
;;     ((a 2) (b 1) (c 3) (d 5))
;; Where a, b,c and d are the units that will make up the messages
;; and the numbers are their corresponding likelihood of occurance.

(define (make-message-inventor
 %message-list
 %grammar
 %min-message-length
 %max-message-length)

  (let ((length1-message-gen
 (make-random-message-generator
  %message-list
   1
   1))
(%rule-set (%grammar 'rule-set)))


    (define (grammar)
      %grammar)


    ;; pick out the rules with the max match in meaning
    ;; among those that have the same lhs
    ;;
```

```
    ;; The fvs-meaning is made by calling
    ;; (make-fv-group (init (random-fv-gen 'get-fv-list)))

    (define (pick-max-match-rules lhs fvs-meaning)
      (let ((rules ((%grammar 'find-rules) lhs))
    (x (return-max-match-num lhs fvs-meaning)))
(list-transform-positive
    rules
  (lambda (r)
    (= x ((%rule-set 'max-#-matching-fvs)
  fvs-meaning
  r))))))


    ;; pick out the rule that is the most expressive, or, can generate
    ;; the most number of meanings among a list of rules

    (define (pick-most-expressive-rule rules)
      (define (pick-aux ls best-rule can-express-n)
(if (null? ls)
    best-rule
    (let* ((r (car ls))
   (x ((%rule-set 'possible-#-meanings) r)))
      (if (> x can-express-n)
  (pick-aux (cdr ls) r x)
  (pick-aux (cdr ls) best-rule can-express-n)))))
      (pick-aux rules '() 0))


    (define (pick-max-match-rule lhs fvs-meaning)
      (let ((rules (pick-max-match-rules lhs fvs-meaning)))
(if (null? rules)
    (error "Can not find best matching rule for invented meaning "
   (fvs-meaning 'print))
    (pick-most-expressive-rule rules))))



    ;; return the max matching number of fvs among those
    ;; rules that have the same lhs

    (define (return-max-match-num lhs fvs-meaning)
      (let ((rules ((%grammar 'find-rules) lhs))
    (the-max 0))
(for-each
```

```
   (lambda (r)
     (let ((x ((%rule-set 'max-#-matching-fvs) fvs-meaning r)))
       (if (> x the-max)
   (set! the-max x)
   '())))
   rules)
the-max))



    ;; This function invents message at places that meaning
    ;; doesn't match the given fvs. It does a one-to-one
    ;; replacement of nonterminals in the rule. So the resulting
    ;; message will have the same length as the template.
    ;; This might be a problem. The invention might be too
    ;; restricted and not random enough.

    (define (fill-in-message the-rule the-fvs-meaning)
      (define (fill-in-aux rule fvs-meaning)
(let ((rhs (rule 'rhs))
      (rhs-len ((rule 'rhs) 'length))
      (invent? (not ((((rule 'meaning) 'fvs) 'subset-of?)
     fvs-meaning))))
  (define (loop i)
    (if (>= i rhs-len)
'()
(let* ((rhs-elm ((rhs 'get-ith) i))
       (is-terminal? ((%grammar 'is-terminal?) rhs-elm))
       (is-non-terminal? ((%grammar 'is-non-terminal?)
  rhs-elm)))
  (cond ((and is-terminal? (not invent?))
 (append (list rhs-elm) (loop (+ i 1))))
((and is-terminal? invent?)
 (append (length1-message-gen 'get-message)
 (loop (+ i 1))))
(is-non-terminal?
 (append
  (fill-in-aux
   (pick-max-match-rule
    (make-lhs (list rhs-elm))
    fvs-meaning)
   fvs-meaning)
  (loop (+ i 1))))
(else
 (begin
```

```
   (%grammar 'print)
   (newline)(display rhs-elm)(newline)
   (rule 'print)
   (error
    "rhs element is
                            neither terminal nor nonterminal")))))))
  (loop 0)))
      (define (contains-non-terminal? ls)
(list-search-positive
    ls
  (lambda (x)
    ((%grammar 'is-non-terminal?) x))))
      (let ((the-msg (fill-in-aux the-rule the-fvs-meaning)))

(if (contains-non-terminal? the-msg)
    (begin
      (newline)
      (display "The message invented contains non-terminal.")
      (newline)
      (display the-msg)(newline)
      (the-rule 'print)(newline)
      (the-fvs-meaning 'print)
      (%grammar 'print)
      (error "non-terminal in invented message")))
(if (> 3 (length the-msg))
    (begin
      (display "The message invented: ")
      (display the-msg)(newline)
      (display
       "The rule that is to be used as
                template to invent the meaning: \n")
      (the-rule 'print)
      (display "the meaning to be invented: \n")
      (the-fvs-meaning 'print)(newline)
      (%grammar 'print)
      (error "message-invention error"))
    the-msg)))


    ;; If the meaning can be generated by the current grammar,
    ;; then, return the rule that generates that meaning.
    ;; else, return #f
    (define (can-generate-meaning? fv-ls)
      (let* ((fvs-meaning (make-fv-group (init fv-ls)))
     (max-match-num (return-max-match-num
```

```
    (make-lhs '(S))
    fvs-meaning)))
(if (equal? max-match-num (fvs-meaning 'length))
    (pick-max-match-rule (make-lhs '(S)) fvs-meaning)
    #f)))


    ;; Generate all possible ways to express a certain meaning, then,
    ;; compare it with the expression given. If match is found,
    ;; then we can parse that message.
    ;; This is probably very inefficient, but it's a quick fix.

    (define (can-parse? msg fv-ls)

      ;; returns a list of ways this meaning can be expressed
      ;; assuming at this point, we already know that the meaning
      ;; can be fully generated by the grammer.

      (define (find-possible-expressions fv-list)
(let* ((fvs-meaning (make-fv-group (init fv-ls)))
       (rules (pick-max-match-rules (make-lhs '(S)) fvs-meaning)))
  (map
   (lambda (r)
     (fill-in-message r fvs-meaning))
   rules)))

      (if (can-generate-meaning? fv-ls)
  (let ((possible-messages (find-possible-expressions fv-ls)))
    (member msg possible-messages))
  #f))



    (define (invent fv-ls)
      (let* ((fvs-meaning (make-fv-group (init fv-ls)))
     (max-match-num (return-max-match-num
    (make-lhs '(S))
    fvs-meaning))
     (total-match-rule (can-generate-meaning? fv-ls)))

 ;; If we can generate the meaning in full, then
 ;; return the message (have to expand the rule with fill-in-msg)
 ;; else, some nonterminals may be appear on the rhs.
(cond (total-match-rule (fill-in-message
 total-match-rule
```

```
fvs-meaning))

      ;; If there is no overlap between the new meaning and
      ;; the meaning that we can generate with our existing S
      ;; rules, then, we need to invent a totally new message.
      ((equal? 0 max-match-num)
       ((make-random-message-generator
%message-list
%min-message-length
%max-message-length)
'get-message))

      ;; Else use the S rule with the max overlap of
      ;; meaning to produce a new message with invention.
      (else (fill-in-message
     (pick-max-match-rule (make-lhs '(S)) fvs-meaning)
     fvs-meaning)))))


    (define (dispatch m)
      (cond ((eq? m 'invent) invent)
    ((eq? m 'can-generate-meaning?) can-generate-meaning?)
    ((eq? m 'can-parse?) can-parse?)
    ((eq? m 'pick-max-match-rule) pick-max-match-rule)
    ((eq? m 'grammar) (grammar))
    (else
     (error "Unknown operation in MAKE-MESSAGE-INVENTOR -- " m))))
    dispatch))




;; Random-Message-Generator generates random strings in the length
;; range requested.
;;
;; ls is in the form of ((a 2) (b 1) (c 4))
;; the numbers specifying the likelihood of each symbol's occurance

(define (make-random-message-generator ls %min-len %max-len)
  (let ((%wheel (make-roulette-wheel ls)))

    (define (get-message)
      (define (get-aux l)
(if (= 0 l)
    '()
```

```
    (cons (%wheel 'get-symbol)
  (get-aux (- l 1)))))
      (if (> %min-len %max-len)
  (error "ERROR: the min-len in MESSAGE GENERATOR is > max-len")
  (let* ((x (+ 1 (- %max-len %min-len)))
 (random-length (+ %min-len (random x))))
    (get-aux random-length))))

    (define (set-max-length! l)
      (begin
(set! %max-len l)
unspecific))

    (define (set-min-length! l)
      (begin
(set! %min-len l)
unspecific))

    (define (dispatch m)
      (cond ((eq? m 'get-message) (get-message))
    ((eq? m 'set-max-length!) set-max-length!)
    ((eq? m 'set-min-length!) set-min-length!)
    (else
     (error
      "Unknown operation in -- RANDOM MESSAGE GENERATOR " m))))
    dispatch))
```

A.6 Meaning Generator

```
(declare (usual-integrations))


;; This function randomly picks a meaning from the meaning-space
;; of the agent

;; Feat-values-ls is of the form:
;;   '((11 (black white red))
;;     (12 (white red))
;;     (21 (black white)))
;;
;; the method 'get-fv-list returns a list of fv lists that
;;  denotes a meaning. such as '((11 white) (12 black) (21 red))

(define (make-random-fv-generator %feat-values-ls)

  (define (get-fv-list)
    (define (get-aux ls)
      (if (null? ls)
  '()
  (let ((feat (first (first ls)))
(values (second (first ls))))
    (cons
     (list feat
   (list-ref values (random (length values))))
     (get-aux (cdr ls))))))
    (get-aux %feat-values-ls))

  (define (dispatch m)
    (cond ((eq? m 'get-fv-list) (get-fv-list))
  (else (error "Unknown Operation -- RANDOM FV GENERATOR " m))))
  dispatch)
```

A.7 Meaning-representation-specific Operations

```
(declare (usual-integrations))

;; Meaning representation specific operations:
;; This are some of the pattern recognition functions that
;; agent designers need to supply to the Language Evolving Module.
;;
;;  Meaning-diff-one-chunk?  and
;;  Meaning-is-strict-subchunk?
;;


(define *filler-id* 100000)
(define *filler-lhs* '())
(define *filler-rhs* '())


;; Returns (fvs-diffs . cat-list-diffs) where
;; fvs-diffs = (fvs1-lacks . fvs2-lakcs) and
;; cat-list-diffs = (cat-list1-lacks . cat-list2-lacks)
(define (meaning-diff m1 m2)
  (let* ((fvs1 (m1 'fvs))
         (fvs2 (m2 'fvs))
         (fvs-diffs ((fvs1 'diff) fvs2))
         (cat-ls-diffs ((m1 'diff-cat-lists) m2)))
    (cons fvs-diffs cat-ls-diffs)))

(define (meaning-is-strict-subchunk? m1 m2)
  (let* ((fvs&cat-diffs (meaning-diff m1 m2))
         (fvs1-lacks (car (car fvs&cat-diffs)))
         (fvs2-lacks (cdr (car fvs&cat-diffs)))
         (cat1-lacks (car (cdr fvs&cat-diffs)))
         (cat2-lacks (cdr (cdr fvs&cat-diffs))))
    (if (and (null? fvs2-lacks)
             (null? cat2-lacks)
             (or (not (null? fvs1-lacks))
  (not (null? cat1-lacks))))
        fvs&cat-diffs
#f)))

(define (fvs-has-overlap? fvs1 fvs2)
  (let ((ls1 (fvs1 'fvs->list)))
    (there-exists?
```

```
ls1
      (lambda (fv)
((fvs2 'has-member?) fv)))))

(define (cat-list-has-overlap? c1 c2)
  (there-exists?
      c1
    (lambda (c)
      (member c c2))))

;; We have to make sure that there is some overlap, and
;; that the two meanings are not completely different. And, meanings
;; are not in subset relation.
;; And, the the two meaning must contain the same features and positions.
;; Returns ( (fv1-lacks . fv2 lacks) . (cat-ls1-lacks . cat-ls2-lacks) )
(define (meaning-diff-one-chunk? rule-set r1 r2)
  (let ((m1 (r1 'meaning))
(m2 (r2 'meaning)))
    (if (and (((m1 'fvs) 'same-positions?) (m2 'fvs))
     (((m1 'fvs) 'same-features?) (m2 'fvs))
     (not (meaning-is-strict-subchunk? m1 m2))
     (not (meaning-is-strict-subchunk? m2 m1))
     (not ((rule-set 'meaning-equal?) r1 r2))
     ; not completely-different
     (or (fvs-has-overlap? (m1 'fvs) (m2 'fvs))
 (cat-list-has-overlap? (m1 'cat-list)
(m2 'cat-list))))
(meaning-diff m1 m2)
#f)))


;; Returns the meaning chunk that m2 has that is missing in m1.
;; m2 has more positions and/or cats
;(define (one-less-meaning-chunk? m1 m2)

;; Returns the number of fv's that the two meanings differ by,
;; convert the category names to number of fv's. The same category
;; name can correspond to several different rules with different
;; number of fvs in them. In this case, we take the average.
(define (meaning-diff-by? rule-set r1 r2)
  (let* ((m1 (r1 'meaning))
 (m2 (r2 'meaning))
 (fvs&cat-diffs (meaning-diff m1 m2))
 (fvs1-lacks (car (car fvs&cat-diffs)))
 (fvs2-lacks (cdr (car fvs&cat-diffs)))
```

```
      (cat-ls1-lacks (car (cdr fvs&cat-diffs)))
      (cat-ls2-lacks (cdr (cdr fvs&cat-diffs)))
      (rule1 (make-rule *filler-id*
                        *filler-lhs*
                        *filler-rhs*
                        (make-meaning (make-fv-group (init '()))
                                      cat-ls1-lacks)))
      (rule2 (make-rule *filler-id*
                        *filler-lhs*
                        *filler-rhs*
                        (make-meaning (make-fv-group (init '()))
                                      cat-ls2-lacks))))
(+ (length fvs1-lacks)
   (length fvs2-lacks)
   ((rule-set 'avg-#-fvs) rule1)
   ((rule-set 'avg-#-fvs) rule2))))
```

A.8 Attribute Grammar

```
(declare (usual-integrations))

(load-option 'hash-table)


;; Functions for creating and manipulating an attribute grammar.
;; One restriction of this implementation is that no recursion
;; is allowed in the grammar.
;; The grammar is composed of terminals, non-terminals and a
;; set of production rules.

(define (make-grammar %terminals %non-terminals rule-set)
  (let ((%rule-set (if (null? rule-set)
       (make-rule-set)
       rule-set))
(%current-cat 0))

    (define (is-terminal? t)
      (member t %terminals))

    (define (are-terminals? ls)
      (for-all?
  ls
is-terminal?))

    ;; if non-terminals is null, we let that indicate
    ;; that the author of the grammar can not pick out
    ;; the non-terminals before hand, they are added and
    ;; deleted as needed in the grammar, as it evoles.
    (define (is-non-terminal? nt)
      (there-exists?
  (%rule-set 'rule-set->list)
(lambda (r)
  (((r 'lhs) 'equal?) (make-lhs (list nt))))))

    (define (get-new-cat-name)
      (set! %current-cat (+ 1 %current-cat))
      (if (is-terminal? %current-cat)
  (get-new-cat-name)
  %current-cat))

    (define (get-new-id)
```

```
      (%rule-set 'get-new-id))

    (define (clear)
      (begin
(set! %current-cat 0)
(%rule-set 'clear)
unspecific))

    (define (rule-set-changed?)
      (%rule-set 'changed?))

    (define (clear-rule-set-change-marker!)
      (%rule-set 'clear-change-marker!))

    (define (get-rule-set)
      %rule-set)

    (define (add-rule! rule)
      ((%rule-set 'add-rule!) rule))

    (define (get-rule the-id)
      ((%rule-set 'get-rule) the-id))

    (define (find-rules the-lhs)
      ((%rule-set 'find-rules) the-lhs))

    (define (delete-rule! rule)
      ((%rule-set 'delete-rule!) rule))

    (define (rule-set-length)
      (%rule-set 'length))

    (define (get-ith-rule i)
      ((%rule-set 'get-ith-rule) i))

    (define (run-substitute-cat! cat-pattern cat-new)
      ((%rule-set 'run-substitute-cat!) cat-pattern cat-new))

    (define (rule-set->list)
      (%rule-set 'rule-set->list))

    (define (cat-contains-cat? cat1 cat2)
      (define (rule-contains-cat? r cat)
(let* ((rhs (r 'rhs))
       (rhs-ls (rhs 'rhs->list))
```

```
       (terms (list-transform-positive rhs-ls is-terminal?))
       (non-terms
(list-transform-negative rhs-ls is-terminal?)))
  (cond ((null? non-terms) #f)
((member cat non-terms) #t)
(else
 (there-exists?
     non-terms
   (lambda (non-term)
     (cat-contains-cat? non-term cat2)))))))
      (let ((rules (find-rules (make-lhs (list cat1)))))
(there-exists?
    rules
  (lambda (r)
    (rule-contains-cat? r cat2)))))


    (define (print-rules)
      (%rule-set 'print))

    (define (print-grammar)
      (newline)
      (display "Terminals:")(newline)
      (display "----------")(newline)
      (display %terminals)(newline)(newline)
      (display "Rule-set:")(newline)
      (display "---------")(newline)
      (%rule-set 'print))

    (define (dispatch m)
      (cond ((eq? m 'is-terminal?) is-terminal?)
    ((eq? m 'are-terminals?) are-terminals?)
    ((eq? m 'is-non-terminal?) is-non-terminal?)
    ((eq? m 'get-new-cat-name) (get-new-cat-name))
    ((eq? m 'get-new-id) (get-new-id))
           ((eq? m 'clear) (clear))
    ((eq? m 'rule-set-changed?) (rule-set-changed?))
    ((eq? m 'clear-rule-set-change-marker!)
     (clear-rule-set-change-marker!))
    ((eq? m 'rule-set) (get-rule-set))
           ((eq? m 'get-rule) get-rule)
           ((eq? m 'add-rule!) add-rule!)
           ((eq? m 'find-rules) find-rules)
           ((eq? m 'delete-rule!) delete-rule!)
           ((eq? m 'length) (rule-set-length))
```

```
            ((eq? m 'get-ith-rule) get-ith-rule)
            ((eq? m 'run-substitute-cat!) run-substitute-cat!)
            ((eq? m 'rule-set->list) (rule-set->list))
    ((eq? m 'cat-contains-cat?) cat-contains-cat?)
            ((eq? m 'print-rules) (print-rules))
    ((eq? m 'print) (print-grammar))
            (else (error "Unknown operation -- GRAMMAR " m))))
    dispatch))



;; Functions for constructing and manipulating the
;; rule-set. This is the main structure of the attribute
;; grammar

(define (make-rule-set . initial-size)
  (let ((%rules (if (null? initial-size)
    (make-eq-hash-table 100)
    (make-eq-hash-table
     (car initial-size))))
(%current-id -1)
(%change-marker #f))

    (define (get-new-id)
      (set! %current-id (+ 1 %current-id))
      %current-id)

    (define (max-id)
      %current-id)

    (define (clear)
      (begin
(hash-table/clear! %rules)
(set! %current-id -1)
(set! %change-marker #f)
unspecific))

    (define (changed?)
      %change-marker)

    (define (clear-change-marker!)
      (begin
(set! %change-marker #f)
unspecific))
```

```
    (define (add-rule! rule)
      (begin
(hash-table/put! %rules (rule 'id) rule)
(set! %change-marker #t)
unspecific))

    (define (get-rule the-id)
      (hash-table/get %rules the-id '()))

    (define (find-rules the-lhs)
      (let ((rules-found '()))
(hash-table/for-each
 %rules
 (lambda (id rule)
   (if (((rule 'lhs) 'equal?) the-lhs)
       (set! rules-found
     (cons rule rules-found)))))
rules-found))

    (define (delete-rule! rule)
      (hash-table/remove! %rules (rule 'id))
      (begin
(set! %change-marker #t)
unspecific))

    (define (rule-set-length)
      (hash-table/count %rules))

    (define (valid-rule-ids)
      (hash-table/key-list %rules))

    (define (run-substitute-cat! cat-pattern cat-new)
      (for-each
       (lambda (r)
 ((r 'substitute-cat!) cat-pattern cat-new))
       (rule-set->list)))

    ;; we don't need to set %change-marker in this case, since
    ;; if any work was done, the procs in run-chunk & run-mergeCat
    ;; would have already made modifications to the rule-set.
    ;; Therefore, setting the %change-marker.
    (define (run-remove-duplicates!)
      (define (remove-if-equal! rule)
(hash-table/for-each
```

```
%rules
(lambda (id r)
  (if (and (not (equal? id (rule 'id)))
   (rule-equal? r rule))
      (hash-table/remove! %rules id)))))
     (let ((key-ls (hash-table/key-list %rules)))
(for-each
 (lambda (key1)
  (let ((pattern-rule (hash-table/get %rules key1 '())))
    (if (not (null? pattern-rule))
 (remove-if-equal! pattern-rule))))
 key-ls)))

   (define (rule-set->list)
     (hash-table/datum-list %rules))


   ;; Since a rule can contain non-terminals and each non-terminal
   ;; may have several different rules with different number of
   ;; feature-values pairs in them, it is useful to get estimates
   ;; on the number of fv pairs a rule contain.

   (define (min-#-fvs rule)
     (define (min-#-fvs-aux meaning)
       (define (min-cat-list-sum cat-ls)
         (if (null? cat-ls)
             0
             (let* ((cat (car cat-ls))
                    (rules (find-rules (make-lhs (list cat)))))
               (+ (apply
                   min
                   (map (lambda (r)
                          (min-#-fvs-aux (r 'meaning)))
                        rules))
                  (min-cat-list-sum (cdr cat-ls))))))
       (+ (meaning 'fvs-length)
          (min-cat-list-sum (meaning 'cat-list))))
     (min-#-fvs-aux (rule 'meaning)))

   (define (max-#-fvs rule)
     (define (max-#-fvs-aux meaning)
       (define (max-cat-list-sum cat-ls)
         (if (null? cat-ls)
             0
             (let* ((cat (car cat-ls))
```

```
     (rules (find-rules (make-lhs (list cat)))))
(+ (apply
                    max
                    (map (lambda (r)
                           (max-#-fvs-aux (r 'meaning)))
                         rules))
                   (max-cat-list-sum (cdr cat-ls))))))
       (+ (meaning 'fvs-length)
          (max-cat-list-sum (meaning 'cat-list))))
      (max-#-fvs-aux (rule 'meaning)))

    (define (avg-#-fvs rule)
      (define (avg-#-fvs-aux meaning)
        (define (avg-cat-list-sum cat-ls)
          (if (null? cat-ls)
              0
              (let* ((cat (car cat-ls))
     (rules (find-rules (make-lhs (list cat)))))
                (+ (average
                    (map (lambda (r)
                           (avg-#-fvs-aux (r 'meaning)))
                         rules))
                   (avg-cat-list-sum (cdr cat-ls))))))
       (+ (meaning 'fvs-length)
          (avg-cat-list-sum (meaning 'cat-list))))
      (avg-#-fvs-aux (rule 'meaning)))


    ;; The maximum number of feature-value pairs that matches the
    ;; fvs given. The number of matching fv pairs could be different
    ;; because, again, the same rule might contain terminals that
    ;; is the lhs of several rules.

    (define (max-#-matching-fvs fvs rule)
      (define (max-aux meaning)
        (define (max-cat-list-sum cat-ls)
          (if (null? cat-ls)
              0
              (let* ((cat (car cat-ls))
     (rules (find-rules (make-lhs (list cat)))))
                (+ (apply
                    max
                    (map (lambda (r)
                           (max-aux (r 'meaning)))
                         rules))
```

```
                    (max-cat-list-sum (cdr cat-ls))))))
(let ((num-good-fvs (if (((meaning 'fvs) 'subset-of?) fvs)
(meaning 'fvs-length)
0)))
  (+ num-good-fvs
     (max-cat-list-sum (meaning 'cat-list)))))
      (max-aux (rule 'meaning)))


    ;; from make-meaning:
    ;; ruturns a list of lists of position/features
    ;; that occurs in this meaning

    (define (get-all-positions rule)
      (define (get-aux cat-ls)
(if (null? cat-ls)
    '()
    (append-map
     (lambda (cat)
       (let ((rules (find-rules (make-lhs (list cat)))))
 (append-map
  (lambda (r)
    (((r 'meaning) 'fvs) 'positions))
  rules)))
     cat-ls)))
      (define (remove-dups ls accum)
(cond ((null? ls) accum)
      ((member (car ls) accum)
       (remove-dups (cdr ls) accum))
      (else (remove-dups
     (cdr ls)
     (cons (car ls) accum)))))
      (let* ((cat-list ((rule 'meaning) 'cat-list))
     (fvs ((rule 'meaning) fvs))
     (ls (get-aux cat-list))
     (pos-with-dup (append (fvs 'positions)
   ls)))
(remove-dups pos-with-dup '())))

    (define (possible-fvs rule)
      (define (do-expand cat-ls)
(if (null? cat-ls)
    '()
    (make-combinations
     (map
```

```
      (lambda (cat)
(let ((rules (find-rules (make-lhs (list cat)))))
  (append-map
   (lambda (r)
     (possible-fvs r))
   rules)))
      cat-ls))))
      (let ((fvs-ls (((rule 'meaning) 'fvs) 'fvs->list))
    (cat-list ((rule 'meaning) 'cat-list)))
(if (null? cat-list)
    (list fvs-ls)
    (make-combinations
     (append
      (list (list fvs-ls))
      (list (do-expand cat-list)))))))

    (define (possible-#-meanings rule)
      (length (possible-fvs rule)))

    (define (meaning-equal? r1 r2)
      (define (convert-to-fvs the-list)
(map (lambda (ls)
       (make-fv-group ls))
     the-list))
      (let ((fvs-ls1 (convert-to-fvs
      (possible-fvs r1)))
    (fvs-ls2 (convert-to-fvs
      (possible-fvs r2))))
(there-exists?
    fvs-ls1
  (lambda (fvs1)
    (there-exists?
fvs-ls2
      (lambda (fvs2)
((fvs1 'equal?) fvs2)))))))

    (define (rule-equal? r1 r2)
      (and (((r1 'lhs) 'equal?) (r2 'lhs))
   (((r1 'rhs) 'equal?) (r2 'rhs))
   (meaning-equal? r1 r2)))


    (define (print-rules)
      (hash-table/for-each
       %rules
```

```
      (lambda (id rule)
 (rule 'print))))

    (define (dispatch m)
      (cond ((eq? m 'get-new-id) (get-new-id))
    ((eq? m 'max-id) (max-id))
    ((eq? m 'clear) (clear))
    ((eq? m 'changed?) (changed?))
    ((eq? m 'clear-change-marker!) (clear-change-marker!))
            ((eq? m 'get-rule) get-rule)
            ((eq? m 'add-rule!) add-rule!)
            ((eq? m 'find-rules) find-rules)
            ((eq? m 'delete-rule!) delete-rule!)
            ((eq? m 'length) (rule-set-length))
    ((eq? m 'valid-rule-ids) (valid-rule-ids))
            ((eq? m 'get-ith-rule) get-ith-rule)
            ((eq? m 'run-substitute-cat!) run-substitute-cat!)
    ((eq? m 'run-remove-duplicates!) (run-remove-duplicates!))
            ((eq? m 'rule-set->list) (rule-set->list))
    ((eq? m 'min-#-fvs) min-#-fvs)
            ((eq? m 'max-#-fvs) max-#-fvs)
            ((eq? m 'avg-#-fvs) avg-#-fvs)
    ((eq? m 'max-#-matching-fvs) max-#-matching-fvs)
    ((eq? m 'get-all-positions) get-all-positions)
    ((eq? m 'possible-fvs) possible-fvs)
    ((eq? m 'possible-#-meanings) possible-#-meanings)
    ((eq? m 'meaning-equal?) meaning-equal?)
    ((eq? m 'rule-equal?) rule-equal?)
            ((eq? m 'print) (print-rules))
            (else (error "Unknown operation -- RULE-SET " m))))
    dispatch))



;; A rule contains a left-hand-side, a right-hand-side,
;; and a set of meaning in the form of feature-value pairs.

(define (make-rule %id %lhs %rhs %meaning)

    (define (get-id) %id)

    (define (get-lhs) %lhs)

    (define (get-rhs) %rhs)
```

```
    (define (get-meaning) %meaning)

    (define (change-lhs new-lhs)
      (make-rule %id new-lhs %rhs %meaning))

    (define (change-lhs! new-lhs)
      (begin
(set! %lhs new-lhs)
unspecific))

    (define (change-rhs new-rhs)
      (make-rule %id %lhs new-rhs %meaning))

    (define (change-rhs! new-rhs)
      (begin
(set! %rhs new-rhs)
unspecific))

    (define (change-meaning m)
      (make-rule %id %lhs %rhs m))

    (define (change-meaning! m)
      (begin
(set! %meaning m)
unspecific))

    (define (substitute-cat cat-pattern cat-new)
      (let ((lhs ((%lhs 'substitute) cat-pattern cat-new))
            (rhs ((%rhs 'substitute-cat) cat-pattern cat-new))
            (m ((%meaning 'substitute-cat) cat-pattern cat-new)))

(if (and (equal? (car (%lhs 'lhs->list)) cat-new)
 (member cat-pattern (%rhs 'rhs->list)))
    (begin (print-rule)
   (display "cat-pattern: ")
   (display cat-pattern)(newline)
   (display "cat-new: ")
   (display cat-new)(newline)
   (error
                    "error: subsituting cat on rhs makes recursion.")))
(if (and (equal? cat-pattern (car (%lhs 'lhs->list)))
 (member cat-new (%rhs 'rhs->list)))
    (begin (print-rule)
   (display "cat-pattern: ")
   (display cat-pattern)(newline)
```

```
  (display "cat-new: ")
  (display cat-new)(newline)
  (error
    "error: subsituting cat on lhs makes recursion.")))
       (make-rule %id
                  lhs
                  rhs
                  m)))

   (define (substitute-cat! cat-pattern cat-new)
     (if (and (equal? (car (%lhs 'lhs->list)) cat-new)
      (member cat-pattern (%rhs 'rhs->list)))
 (begin (print-rule)
(display "cat-pattern: ")
(display cat-pattern)(newline)
(display "cat-new: ")
(display cat-new)(newline)
(error
                  "error: subsituting-cat! on rhs makes recursion.")))
     (if (and (equal? (car (%lhs 'lhs->list)) cat-pattern)
      (member cat-new (%rhs 'rhs->list)))
 (begin (print-rule)
(display "cat-pattern: ")
(display cat-pattern)(newline)
(display "cat-new: ")
(display cat-new)(newline)
(error
                 "error: subsituting-cat! on lhs makes recursion.")))
     ((%lhs 'substitute!) cat-pattern cat-new)
     ((%rhs 'substitute-cat!) cat-pattern cat-new)
     ((%meaning 'substitute-cat!) cat-pattern cat-new))


   (define (print-rule)
     (newline)
     (display "#")
     (display %id)
     (newline)
     (%lhs 'print)
     (display " -> ")
     (%rhs 'print)
     (newline)
     (%meaning 'print))

   (define (dispatch m)
```

```
      (cond ((eq? m 'id) (get-id))
            ((eq? m 'lhs) (get-lhs))
            ((eq? m 'rhs) (get-rhs))
            ((eq? m 'meaning) (get-meaning))
            ((eq? m 'change-lhs) change-lhs)
            ((eq? m 'change-lhs!) change-lhs!)
            ((eq? m 'change-rhs) change-rhs)
            ((eq? m 'change-rhs!) change-rhs!)
            ((eq? m 'change-meaning) change-meaning)
            ((eq? m 'change-meaning!) change-meaning!)
            ((eq? m 'substitute-cat) substitute-cat)
            ((eq? m 'substitute-cat!) substitute-cat!)
            ((eq? m 'print) (print-rule))
            (else (error "Unknown operation -- RULE " m))))

    dispatch)



;; Lhs will be a list of non-terminals although the grammar
;; we are using now only have one non-terminal

(define (make-lhs %ls)

    (define (lhs-length)
      (length %ls))

    (define (lhs->list)
      %ls)

    (define (lhs-equal? lhs)
      (equal? %ls (lhs 'lhs->list)))

    (define (substitute old new)
      (make-lhs
       (map (lambda (cat)
              (if (equal? cat old)
                  new
                  cat))
            %ls)))

    (define (substitute! old new)
      (begin
(set! %ls
      (map (lambda (cat)
```

```
     (if (equal? cat old)
 new
 cat))
   %ls))
unspecific))

    (define (print)
      (print-list %ls))

    (define (dispatch m)
      (cond ((eq? m 'length) (lhs-length))
            ((eq? m 'lhs->list) (lhs->list))
            ((eq? m 'equal?) lhs-equal?)
            ((eq? m 'substitute) substitute)
            ((eq? m 'substitute!) substitute!)
            ((eq? m 'print) (print))))
    dispatch)



;; The rhs is made up of terminals and non-terminals.

(define (make-rhs %ls)

    (define (contains-cat? cat)
      (member cat %ls))

    (define (rhs-length)
      (length %ls))

    (define (rhs->string)
      (list->string %ls))

    (define (rhs->list)
      %ls)

    (define (is-empty?)
      (null? %ls))

    (define (get-ith i)
      (list-ref %ls i))

    (define (rhs-equal? rhs)
      (equal? %ls (rhs 'rhs->list)))
```

```
    (define (substitute-cat old new)
      (make-rhs
       (map (lambda (cat)
              (if (equal? cat old)
                  new
                  cat))
            %ls)))

    (define (substitute-cat! old new)
      (begin
(set! %ls
      (map (lambda (cat)
     (if (equal? cat old)
 new
 cat))
   %ls))
unspecific))

    ;; Only substitutes the last pattern encountered in the rhs.
    ;; ex. if rhs is (1 2 0 0 0 3 0 0 0) subtitutes (1 1 1)
    ;; for 0 0 0 will get: (1 2 0 0 0 3 1 1 1)
    (define (substitute-cats old-cats new-cats)
      (define (substitute-aux rhs-ls)
        (if (not (substring-of? old-cats (cdr rhs-ls)))
            (append new-cats
                    (list-tail rhs-ls (length old-cats)))
            (append (list (car rhs-ls))
                    (substitute-aux (cdr rhs-ls)))))

      (if (not (substring-of? old-cats %ls))
          (make-rhs %ls)
          (make-rhs (substitute-aux %ls))))

    (define (substitute-cats! old-cats new-cats)
      (define (substitute-aux rhs-ls)
        (if (not (substring-of? old-cats (cdr rhs-ls)))
            (append new-cats
                    (list-tail rhs-ls (length old-cats)))
            (append (list (car rhs-ls))
                    (substitute-aux (cdr rhs-ls)))))
      (if (substring-of? old-cats %ls)
          (begin
    (set! %ls (substitute-aux %ls))
    unspecific)))
```

```
  ;; they have to be in strictly substring relationship.
  (define (is-substring? rhs)
    (if (equal? (rhs 'length) (rhs-length))
#f
(substring-of? %ls (rhs 'rhs->list))))

  ;; The rhs cats has to be in the same order
  ;; Returns cats in as a pair or #f if they don't diff
  ;; by only one pair of cats.
  (define (diff-one-pair-cats? rhs)
    (define (diff-aux ls1 ls2 diff-ls)
      (cond ((and (null? ls1) (null? ls2)) diff-ls)
            ((null? ls1) (append ls2 diff-ls))
            ((null? ls2) (append ls1 diff-ls))
            ((equal? (car ls1) (car ls2))
             (diff-aux (cdr ls1) (cdr ls2) diff-ls))
            (else (diff-aux (cdr ls1)
                            (cdr ls2)
                            (cons (list (car ls1) (car ls2))
                                  diff-ls)))))
    (let* ((rhs1 %ls)
           (rhs2 (rhs 'rhs->list))
           (the-diff (diff-aux rhs1 rhs2 '())))
      (cond ((not (equal? (length rhs1) (length rhs2)))
             #f)
            ((equal? 1 (length the-diff))
             (car the-diff))
            (else #f))))

  ;; Returns x1 x2 as a pair, or '() if no chunking
  ;; ex  (find-chunks '(1 2 3 4 5) '(1 2 3 6) 1 3)
  ;;  ==> ((4 5) (6))
  ;; ex  (find-chunks '(5 5 1 2 3) '(0 0 0 1 2 3) 1 3)
  ;;  ==> ((5 5) (0 0 0))
  ;; If rhs1 and rhs2 diff by more than 2 substrings,
  ;; then, no chunking occurs. ex '(1 2 2 2 4) '(5 2 2 2)
  ;; They diff by 3 substrings
  (define (find-chunks rhs min_ck_len max_ck_len)
    (define (good-chunks? x1_len x2_len min_len max_len)
      (cond ((and (= x1_len 1) (= x2_len 1)) #f)
            ((and (and (>= x1_len min_len) (<= x1_len max_len))
                  (and (>= x2_len min_len) (<= x2_len max_len)))
             #t)
            (else #f)))
    (let* ((rhs1 %ls)
```

```
        (rhs2 (rhs 'rhs->list))
        (rhs1_len (length rhs1))
        (rhs2_len (length rhs2))
        (str1 (list->string rhs1))
        (str2 (list->string rhs2))
        (f_common_len (string-match-forward str1 str2))
        (b_common_len (string-match-backward str1 str2))
        (f_x1_len (- rhs1_len f_common_len))
        (f_x2_len (- rhs2_len f_common_len))
        (b_x1_len (- rhs1_len b_common_len))
        (b_x2_len (- rhs2_len b_common_len))
        (f_x1 (list-tail rhs1 f_common_len))
        (f_x2 (list-tail rhs2 f_common_len))
        (b_x1 (list-head rhs1 (- rhs1_len b_common_len)))
        (b_x2 (list-head rhs2 (- rhs2_len b_common_len)))
        (f_good
  (good-chunks? f_x1_len f_x2_len min_ck_len max_ck_len))
        (b_good
         (good-chunks? b_x1_len b_x2_len min_ck_len max_ck_len)))
    (cond ((and (>= f_common_len b_common_len)
                f_good
                (not (equal? 0 f_common_len)))
           (cons f_x1 (list f_x2)))
          ((and b_good
                (not (equal? 0 b_common_len)))
           (cons b_x1 (list b_x2)))
          ((and f_good
                (not (equal? 0 f_common_len)))
           (cons f_x1 (list f_x2)))
          (else '()))))

(define (print)
  (print-list %ls))

(define (dispatch m)
  (cond ((eq? m 'contains-cat?) contains-cat?)
        ((eq? m 'length) (rhs-length))
        ((eq? m 'rhs->string) (rhs->string))
        ((eq? m 'rhs->list) (rhs->list))
((eq? m 'empty?) (is-empty?))
        ((eq? m 'equal?) rhs-equal?)
((eq? m 'get-ith) get-ith)
        ((eq? m 'substitute-cat) substitute-cat)
        ((eq? m 'substitute-cat!) substitute-cat!)
        ((eq? m 'substitute-cats) substitute-cats)
```

```
            ((eq? m 'substitute-cats!) substitute-cats!)
            ((eq? m 'substring?) is-substring?)
            ((eq? m 'diff-one-pair-cats?) diff-one-pair-cats?)
            ((eq? m 'find-chunks) find-chunks)
            ((eq? m 'print) (print))
            (else (error "Unknown operation -- RHS " m))))
    dispatch)



(define (make-meaning %fvs . cat-list)
  (let ((%cat-list (if (null? cat-list)
                       '()
                       (car cat-list))))

    (define (get-fvs)
      %fvs)

    (define (get-cat-list)
      %cat-list)

    (define (first-cat)
      (first %cat-list))

    (define (rest-cats)
      (cdr %cat-list))

    (define (cat-list-empty?)
      (null? %cat-list))

    (define (cat-list-length)
      (length %cat-list))

    (define (add-cats cats)
      (make-meaning %fvs
                    (append %cat-list cats)))

    (define (add-cats! cats)
      (begin
(set! %cat-list (append %cat-list cats))
unspecific))

    (define (add-cats-if-lack! cats)
      (for-each
       (lambda (cat)
```

```
 (if (not (member cat %cat-list))
     (set! %cat-list (append %cat-list (list cat)))))
       cats))

    (define (delete-cats ls)
      (let ((cat-ls (list-transform-negative
                       %cat-list
                      (lambda (cat)
                        (member cat ls)))))
        (make-meaning %fvs cat-ls)))

    (define (delete-cats! ls)
      (begin
(set! %cat-list
      (list-transform-negative
  %cat-list
(lambda (cat)
  (member cat ls))))
unspecific))

    (define (add-fv fv)
      (make-meaning
       ((%fvs 'add-fv) fv)
       %cat-list))

    (define (add-fv! fv)
      ((%fvs 'add-fv!) fv))

    (define (add-fvs fvs)
      (make-meaning
       ((%fvs 'add-fvs) fvs)
       %cat-list))

    (define (add-fvs! fvs)
      ((%fvs 'add-fvs!) fvs))

    (define (delete-fv fv)
      (make-meaning
       ((%fvs 'delete-fv) fv)
       %cat-list))

    (define (delete-fv! fv)
      ((%fvs 'delete-fv!) fv))

    (define (delete-fvs fvs)
```

```
      (make-meaning
       ((%fvs 'delete-fvs) fvs)
       %cat-list))

    (define (delete-fvs! fvs)
      ((%fvs 'delete-fvs!) fvs))

    (define (delete-sub-meaning m)
      (let ((fvs (m 'fvs))
            (cat-ls (m 'cat-list)))
        (make-meaning
         ((%fvs 'delete-fvs) fvs)
         ((delete-cats cat-ls) 'cat-list))))

    (define (delete-sub-meaning! m)
      (let ((fvs (m 'fvs))
            (cat-ls (m 'cat-list)))
        ((%fvs 'delete-fvs!) fvs)
        (delete-cats! cat-ls)))

    (define (fvs-length)
      (%fvs 'length))

    (define (cat-list-equal? meaning)
      (lists-same? %cat-list (meaning 'cat-list)))

    ;; returns a new-meaning
    (define (substitute-cat cat-old cat-new)
      (let ((cat-ls (map (lambda (cat)
                           (if (equal? cat cat-old)
                               cat-new
                               cat))
                         %cat-list)))
        (make-meaning %fvs cat-ls)))

    (define (substitute-cat! cat-old cat-new)
      (begin
(set! %cat-list
      (map (lambda (cat)
     (if (equal? cat cat-old)
 cat-new
 cat))
   %cat-list))
unspecific))
```

```
; returns a cons pair of lists, the first holds what
; cat-lists lacks and the second holds what ls lacks.
(define (diff-cat-lists meaning)
  (diff-lists %cat-list (meaning 'cat-list)))

(define (diff-fvs meaning)
  ((%fvs 'diff) (meaning 'fvs)))

(define (diff meaning)
  (cons (diff-lists %cat-list (meaning 'cat-list))
        ((%fvs 'diff) (meaning 'fvs))))

(define (print)
  (%fvs 'print)
  (display %cat-list)
  (newline))

(define (dispatch m)
  (cond ((eq? m 'fvs) (get-fvs))
        ((eq? m 'cat-list) (get-cat-list))
        ((eq? m 'first-cat) (first-cat))
        ((eq? m 'rest-cats) (rest-cats))
        ((eq? m 'cat-list-empty?) (cat-list-empty?))
        ((eq? m 'cat-list-length) (cat-list-length))
        ((eq? m 'add-cats) add-cats)
        ((eq? m 'add-cats!) add-cats!)
((eq? m 'add-cats-if-lack!) add-cats-if-lack!)
        ((eq? m 'delete-cats) delete-cats)
        ((eq? m 'delete-cats!) delete-cats!)
        ((eq? m 'add-fv) add-fv)
        ((eq? m 'add-fv!) add-fv!)
        ((eq? m 'add-fvs) add-fvs)
        ((eq? m 'add-fvs!) add-fvs!)
        ((eq? m 'delete-fv) delete-fv)
        ((eq? m 'delete-fv!) delete-fv!)
        ((eq? m 'delete-fvs) delete-fvs)
        ((eq? m 'delete-fvs!) delete-fvs!)
        ((eq? m 'delete-sub-meaning) delete-sub-meaning)
        ((eq? m 'delete-sub-meaning!) delete-sub-meaning!)
        ((eq? m 'fvs-length) (fvs-length))
        ((eq? m 'cat-list-equal?) cat-list-equal?)
        ((eq? m 'substitute-cat) substitute-cat)
        ((eq? m 'substitute-cat!) substitute-cat!)
        ((eq? m 'diff-cat-lists) diff-cat-lists)
        ((eq? m 'diff-fvs) diff-fvs)
```

```
            ((eq? m 'diff) diff)
            ((eq? m 'print) (print))
            (else (error "Unknown operation -- MEANING " m))))
    dispatch))



;; Make-fv-group takes a list of ORDERED fv pairs, user should
;; enter the pairs in order.
;; (define fvs (make-fv-group (init '((11 black)
;;                                    (12 white)))))

(define (make-fv-group %fvs)

    (define (the-first)
      (first %fvs))

    (define (the-rest)
      (make-fv-group (cdr %fvs)))

    (define (fvs-length)
      (length %fvs))

    (define (fvs->list)
      %fvs)

    ;; returns a list of all the positions
    ;; that occur in this fv-group
    (define (positions)
      (define (positions-aux ls)
(if (null? ls)
    '()
    (cons ((car ls) 'position)
  (positions-aux (cdr ls)))))
      (positions-aux %fvs))

    (define (get-fv-at-position pos)
      (define (get-fv-aux fv-ls)
(cond ((null? fv-ls) '())
      ((equal? pos ((car fv-ls) 'position))
       (car fv-ls))
      (else (get-fv-aux (cdr fv-ls)))))
      (get-fv-aux %fvs))

    (define (add-fv fv)
```

```
      (make-fv-group
       (append %fvs (list fv))))

    (define (add-fv! fv)
      (begin
(set! %fvs (append %fvs (list fv)))
unspecific))

    (define (add-fvs fvs)
      (make-fv-group
       (append %fvs (fvs 'fvs->list))))

    (define (add-fvs! fvs)
      (begin
(set! %fvs (append %fvs (fvs 'fvs->list)))
unspecific))

    (define (has-member? fv)
      (there-exists?
          %fvs
        (lambda (a-fv)
          ((fv 'equal?) a-fv))))

    (define (strict-subset-of? fvs)
      (if (>= (length %fvs) (fvs 'length))
          #f
          (for-all?
              %fvs
            (lambda (fv)
              ((fvs 'has-member?) fv)))))

    ; whether %fvs is a subset of fvs
    (define (subset-of? fvs)
      (if (> (length %fvs) (fvs 'length))
          #f
          (for-all?
              %fvs
            (lambda (fv)
              ((fvs 'has-member?) fv)))))

    (define (delete-fv fv)
      (make-fv-group
       (list-transform-negative
           %fvs
         (lambda (a-fv)
```

```
      ((a-fv 'equal?) fv)))))

(define (delete-fv! fv)
  (set! %fvs
        (list-transform-negative
            %fvs
          (lambda (a-fv)
            ((a-fv 'equal?) fv)))))

(define (delete-fvs fvs)
  (make-fv-group
   (list-transform-negative
       %fvs
     (lambda (fv)
       ((fvs 'has-member?) fv)))))

(define (delete-fvs! fvs)
  (set! %fvs
        (list-transform-negative
            %fvs
          (lambda (fv)
            ((fvs 'has-member?) fv)))))

; returns a cons pair of lists, the first holds
; what %fvs lacks, the second holds what fvs lacks.
(define (diff fvs)
  (define (eq-test fv1 fv2)
    ((fv1 'equal?) fv2))
  (diff-lists %fvs (fvs 'fvs->list) eq-test))

(define (empty?)
  (null? %fvs))

(define (same-positions? fvs)
  (define (same-positions-aux fvs1 fvs2)
    (cond ((fvs1 'empty?) #t)
          ((not (((fvs1 'first) 'position-equal?)
                 (fvs2 'first)))
           #f)
          (else (same-positions-aux (fvs1 'rest)
                                    (fvs2 'rest)))))
  (let ((fvs1 (sort-fvs))
        (fvs2 (fvs 'sort)))
    (if (equal? (fvs1 'length) (fvs2 'length))
        (same-positions-aux fvs1 fvs2)
```

```
          #f)))

    (define (same-features? fvs)
      (define (same-features-aux fvs1 fvs2)
        (cond ((fvs1 'empty?) #t)
              ((not (((fvs1 'first) 'feature-equal?)
                     (fvs2 'first)))
               #f)
              (else (same-features-aux (fvs1 'rest)
                                       (fvs2 'rest)))))
      (let ((fvs1 (sort-fvs))
            (fvs2 (fvs 'sort)))
        (if (equal? (fvs1 'length) (fvs2 'length))
            (same-features-aux fvs1 fvs2)
            #f)))

    ;; same positions & same values
    (define (fvs-equal? fvs)
      (define (equal-aux fvs1 fvs2)
        (cond ((fvs1 'empty?) #t)
              ((not (((fvs1 'first) 'equal?)
                     (fvs2 'first)))
               #f)
              (else (equal-aux (fvs1 'rest)
                               (fvs2 'rest)))))
      (let ((fvs1 (sort-fvs))
            (fvs2 (fvs 'sort)))
        (if (equal? (fvs1 'length) (fvs2 'length))
            (equal-aux fvs1 fvs2)
            #f)))

    (define (sort-fvs)
      (make-fv-group
       (sort %fvs
             (lambda (fv1 fv2)
               (< (fv1 'position)
                  (fv2 'position))))))

    (define (sort-fvs!)
      (begin
(set! %fvs (sort %fvs
 (lambda (fv1 fv2)
   (< (fv1 'position)
      (fv2 'position)))))
unspecific))
```

```
    (define (print)
      (for-each (lambda (fv)
                  (fv 'print))
                %fvs))

    (define (dispatch m)
      (cond ((eq? m 'first) (the-first))
            ((eq? m 'rest) (the-rest))
            ((eq? m 'fvs->list) (fvs->list))
    ((eq? m 'positions) (positions))
    ((eq? m 'get-fv-at-position) get-fv-at-position)
            ((eq? m 'add-fv) add-fv)
            ((eq? m 'add-fv!) add-fv!)
            ((eq? m 'add-fvs) add-fvs)
            ((eq? m 'add-fvs!) add-fvs!)
            ((eq? m 'length) (fvs-length))
            ((eq? m 'has-member?) has-member?)
            ((eq? m 'strict-subset-of?) strict-subset-of?)
            ((eq? m 'subset-of?) subset-of?)
            ((eq? m 'delete-fv) delete-fv)
            ((eq? m 'delete-fv!) delete-fv!)
            ((eq? m 'delete-fvs) delete-fvs)
            ((eq? m 'delete-fvs!) delete-fvs!)
            ((eq? m 'diff) diff)
            ((eq? m 'empty?) (empty?))
            ((eq? m 'same-positions?) same-positions?)
            ((eq? m 'same-features?) same-features?)
            ((eq? m 'equal?) fvs-equal?)
            ((eq? m 'sort) (sort-fvs))
            ((eq? m 'sort!) (sort-fvs!))
            ((eq? m 'print) (print))
            (else (error "Unknown operation -- FV-GROUP " m))))
    dispatch)



(define (make-fv %position %feature %value)

    (define (get-feature)
      %feature)

    (define (get-value)
      %value)
```

```
    (define (get-position)
      %position)

    (define (position-equal? fv)
      (equal? %position (fv 'position)))

    (define (feature-equal? fv)
      (equal? %feature (fv 'feature)))

    (define (value-equal? fv)
      (equal? %value (fv 'value)))

    (define (fv-equal? fv)
      (and (equal? %position (fv 'position))
           (equal? %feature (fv 'feature))
           (equal? %value (fv 'value))))

    (define (change-feature new-feat)
      (make-fv %position new-feat %value))

    (define (change-feature! new-feat)
      (begin
(set! %feature new-feat)
unspecific))

    (define (change-value new-val)
      (make-fv %position %feature new-val))

    (define (change-value! new-val)
      (begin
(set! %value new-val)
unspecific))

    (define (print)
      (display "  ")
      (display %position)
      (display "  ")
      (display %feature)
      (display " = ")
      (display %value)
      (newline))

    (define (dispatch m)
      (cond ((eq? m 'feature) (get-feature))
            ((eq? m 'value) (get-value))
```

```
        ((eq? m 'position) (get-position))
        ((eq? m 'position-equal?) position-equal?)
        ((eq? m 'feature-equal?) feature-equal?)
        ((eq? m 'value-equal?) value-equal?)
        ((eq? m 'equal?) fv-equal?)
        ((eq? m 'change-feature) change-feature)
        ((eq? m 'change-feature!) change-feature!)
        ((eq? m 'change-value) change-value)
        ((eq? m 'change-value!) change-value!)
        ((eq? m 'print) (print))
        (else (error "Unknown operation -- FV " m))))
dispatch)
```

A.9 Helper Procedures

```
(declare (usual-integrations))

;; This file contains miscellaneous helper functions used
;; throughout the main program.


;; Initializing fv-group when given a fv-list, used in Make-Rule
;; Returns a list of ((position feature) value)'s
;; map was not used to ensure order of the output list
;; is the same as the order of the one given.

(define (init fv-list . start-index)
  (let ((i (if (null? start-index)
       0
       (car start-index))))
    (define (init-loop ls count)
      (if (null? ls)
  '()
  (cons (make-fv count (first (car ls)) (second (car ls)))
(init-loop (cdr ls) (+ 1 count)))))
    (init-loop fv-list i)))


(define (one-in-n-chance n)
  (if (<= n 0)
      (error "n needs to be a positive number in one-in-n-chance")
      (let ((i (random n)))
(equal? i 0))))


;; Randomly picks an element from ls
(define (pick-random ls)
  (let ((len (length ls)))
    (list-ref ls (random len))))


;; the-ls is a list of lists(note, not pairs)
;; the likelihood have to be specified as integers with
;; no .0 following them. Else random number gen will
;; generate diff numbers.
;;(define r (make-roulette-wheel '((a 1) (b 4) (c 3))
```

```
(define (make-roulette-wheel the-ls)
  (let* ((%wheel '()))

    (define (init)
      (define (init-aux ls cumm)
        (if (null? ls)
            '()
            (cons (cons (car (car ls))
                        (+ cumm (cadr (car ls))))
                  (init-aux (cdr ls)
                            (+ cumm (cadr (car ls)))))))
      (set! %wheel (init-aux the-ls 0)))


    ; (random x) generated numbers between
    ; 0(inclusive) and x(exclusive). If x is
    ; an integer, the number returned are ints.
    ; So (random 3) will give 0, 1, or 2
    ; A roulette of ((a 2) (b 4) (c 1)) will be
    ; (2 6 7), x will be 8(so 7 can be generated).
    ; so 0,1 will return a
    ;    2,3,4,5 will return b
    ;    6 will return c

    (define (get)
      (let* ((l (length %wheel))
             (the-max (cdr (list-ref %wheel (- l 1))))
             (r (random the-max)))
        (define (get-aux ls)
          (if (> (cdr (car ls)) r)
              (car (car ls))
              (get-aux (cdr ls))))
        (get-aux %wheel)))

    (define (dispatch m)
      (cond ((eq? m 'get-symbol) (get))
            (else (error "Unknown operation -- ROULETTE " m))))
    (init)
    dispatch))


;; (fill-list 1 5) -> (1 2 3 4 5)
(define (fill-list min max)
  (if (>= min max)
      min
```

```
      (cons min (fill-list (+ min 1) max))))


;; delete a single instance of the element from the list
;; (delete-one 2 '(1 2 3 2 2 2)) -> (1 3 2 2 2)
(define (delete-one x the-list . eq-test)
  (let ((equal-test (if (null? eq-test)
                        equal?
                        (car eq-test))))
    (define (del-aux ls ls-head)
      (cond ((null? ls) ls-head)
            ((equal-test (car ls) x)
             (append ls-head (cdr ls)))
            (else (del-aux (cdr ls)
                           (cons (car ls) ls-head)))))
    (del-aux the-list '())))

(define (delete-one-instance del-ls ls . eq-test)
  (let ((equal-test (if (null? eq-test) equal? (car eq-test))))
    (define (delete-aux delete-ls the-ls)
      (if (null? delete-ls)
          the-ls
          (delete-aux (cdr delete-ls)
                      (delete-one (car delete-ls)
                                  the-ls
                                  equal-test))))
    (delete-aux del-ls ls)))


;; Returns what elements that occurs in list2 but do not
;; in list1. It takes into account of duplications.
;; ex. (ls1-lacks '(1 2 3) '(1 1 2 3)) -> (1)
(define (ls1-lacks list1 list2 . eq-test)
  (let ((equal-test (if (null? eq-test)
                        equal?
                        (car eq-test))))
    (define (lack-aux ls1 ls2 diff)
      (if (null? ls2)
          diff
          (let* ((x (car ls2))
                 (occurs (there-exists?
                          ls1
                          (lambda (elem)
                            (equal-test x elem)))))
            (if occurs
```

```
                (lack-aux (delete-one-instance (list x) ls1 equal-test)
                          (cdr ls2)
                          diff)
                (lack-aux ls1
                          (cdr ls2)
                          (append diff (list x)))))))
    (lack-aux list1 list2 '())))

;; returns a pair of lists, (lack1 lack2)
;; lack1 is what ls2 had that ls1 lacked.
(define (diff-lists ls1 ls2 . eq-test)
  (if (null? eq-test)
      (cons (ls1-lacks ls1 ls2)
            (ls1-lacks ls2 ls1))
      (cons (ls1-lacks ls1 ls2 (car eq-test))
            (ls1-lacks ls2 ls1 (car eq-test)))))

;; True if ls1 and ls2 contains all the same elements.
;; Handles this situation which member testing alone
;; would miss: (1 1 2 3) (1 2 2 3)
(define (lists-same? ls1 ls2)
  (cond ((not (equal? (length ls1)
                      (length ls2)))
         #f)
        ((null? ls1) #t)
        ((member (car ls1) ls2)
         (lists-same? (delete (car ls1) ls1)
                      (delete (car ls1) ls2)))
        (else #f)))


;; The scheme proc substring? doesn't work on examples such as
;;  (4 4 4) (1 2 3 4 4 4) after they are converted
;;  into strings from lists of course.
;; This proc doesn't handle nested lists
(define (substring-of? pattern-ls ls)  ;;; the args are lists
  (let ((pattern (list->string pattern-ls))
        (str (list->string ls)))
    (if (null? pattern-ls)
#t
((substring-search-maker pattern) str))))

;; prints out each element of the list, flatten out sublists.
;; puts a space between each element
;; ex. (print-list (list 0 (list 1 2))) => 0 1 2
```

```
(define (print-list ls)
  (cond ((null? ls) '())
        ((list? (car ls)) (print-list (car ls)))
        (else (display (car ls))
              (display " ")
              (print-list (cdr ls)))))

(define (average ls)
  (if (null? ls)
      0
      (/ (apply + ls)
         (length ls))))

;; Make combinations 2
;; Takes 2 lists of lists and makes lists that are cominbations
;; of those lists:
;; ( ((a b c) (d))  ((u v) (w) (x y z)) )
;; => ((a b c u v) (a b c w) (a b c x y z)
;;     (d u v) (d w) (d x y z))
(define (make-combinations-2 list1 list2)
  (define (combine-elements ls)
    (if (null? ls)
'()
(append (car ls)
(combine-elements (cdr ls)))))
  (let ((lists (map
  (lambda (ls1)
    (map
     (lambda (ls2)
       (append ls1 ls2))
     list2))
  list1)))
    (combine-elements lists)))

;; A more general version that takes list of lists to be
;; made combinations of, rather than just 2.
(define (make-combinations list-of-lists)
  (cond ((null? list-of-lists) '())
((null? (cdr list-of-lists)) (car list-of-lists))
(else
 (make-combinations
  (cons (make-combinations-2 (first list-of-lists)
     (second list-of-lists))
(cdr (cdr list-of-lists)))))))
```

```
;;;------- Bellow is a borrowed function on substring search ---------
; FILE "substr.scm"
; IMPLEMENTS Substring search
; AUTHOR Ken Dickey
; DATE 1991 August 6
; LAST UPDATED

; NOTES
; Based on "A Very Fast Substring Search Algorithm",
;;      Daniel M. Sunday,
; CACM v33, #8, August 1990.
;

;; Gambit-specific compile options

;(##declare
;  (ieee-scheme)
;  (standard-bindings)
;  (lambda-lift)
;  (block)
;  (fixnum))
;;
;; SUBSTRING-SEARCH-MAKER takes a string (the "pattern") and
;; returns a function
;; which takes a string (the "target") and either returns
;; #f or the index in
;; the target in which the pattern first occurs as a substring.
;;
;; E.g.: ((substring-search-maker "test") "This is a test string")
;;       -> 10
;;       ((substring-search-maker "test") "This is a text string")
;;       -> #f

(define (SUBSTRING-SEARCH-MAKER pattern-string)

  (define NUM-CHARS-IN-CHARSET 256)
  ;; Update this, e.g. for ISO Latin 1

  (define (BUILD-SHIFT-VECTOR pattern-string)
    (let* ( (pat-len (string-length pattern-string))
       (shift-vec (make-vector num-chars-in-charset
                               (+ pat-len 1)))
    (max-pat-index (- pat-len 1))
          )
```

```
      (let loop ( (index 0) )
        (vector-set! shift-vec
     (char->integer (string-ref pattern-string index))
     (- pat-len index)
        )
(if (< index max-pat-index)
    (loop (+ index 1))
    shift-vec)
  ) ) )

  (let ( (shift-vec (build-shift-vector pattern-string))
 (pat-len   (string-length pattern-string))
       )

   (lambda (target-string)

      (let* ( (tar-len (string-length target-string))
      (max-tar-index (- tar-len 1))
      (max-pat-index (- pat-len 1))
            )
(let outer ( (start-index 0) )
           (if (> (+ pat-len start-index) tar-len)
       #f
       (let inner ( (p-ind 0) (t-ind start-index) )
          (cond
      ((> p-ind max-pat-index)  ; nothing left to check
    #f        ; fail
   )
   ((char=? (string-ref pattern-string p-ind)
      (string-ref target-string  t-ind))
    (if (= p-ind max-pat-index)
        start-index
                        (inner (+ p-ind 1) (+ t-ind 1))
                 )
                )
                ((> (+ pat-len start-index) max-tar-index) #f) ; fail
   (else
     (outer
(+ start-index
 (vector-ref shift-vec
             (char->integer
     (string-ref target-string
     (+ start-index pat-len)
                ) )     ) )          )    )
               ) ; end-cond
```

```
            ) ) )
    )   ) ; end-lambda
) )
```